The Dissertation Committee for Min Kyu Jeong
certifies that this is the approved version of the following dissertation:

Core-characteristic-aware Off-chip Memory Management in a Multicore System-on-Chip

Committee:

Mattan Erez, Supervisor

Lizy K. John

Derek Chiou

Calvin Lin

Michael J. Schulte

Core-characteristic-aware Off-chip Memory Management in a Multicore System-on-Chip

by

Min Kyu Jeong, B.S.C.S.&E.; M.S.E

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2012

To my loving parents,

Gui Soo Jeong and Youn Sun Min.

Acknowledgments

First, I thank my great advisor Mattan Erez. Mattan patiently guided me through many turns during my time in a graduate school. He was a caring, hard-working, supportive, open-minded, yet rigorous advisor who I was able to turn to freely any time I need an advice. I was fortunate to have him as my advisor.

I also thank Doe Hyun Yoon for the opportunity to collaborate with him. He was a role model in our research group whose strong self-motivation and high quality research inspired me. The main idea of this dissertation was developed from the discussions I had with him.

I thank Lizy John, Derek Chiou, Calvin Lin, and Michael Schulte for serving on my dissertation committee and providing me valuable comments on this dissertation.

I would like to thank Nagi Aboulenein and Hemant Rotithor at Intel Portland. I learned a lot about memory systems while I was interning with them, based on which my research interests in memory system started. I am also grateful to Nigel Paver and Chander Sudanthi at ARM Austin for their support to my research. They gave me a very fortunate opportunity to experiment my idea with valuable resources and the study resulted in a chapter in this dissertation.

I thank Ikhwan Lee and Sangmin Lee for their friendship. It was their brotherly company that kept me through my time in Austin. I am grateful for their tolerance with me and genuine life advices they gave me by examples.

I am grateful to Dam Sunwoo for his help at every major milestone in my graduate school career. He helped me getting an internship at ARM, wrote my first paper together, and gave valuable comments on my proposal and final defense practices.

I would also like to acknowledge the old members of LPH group: Mike Sullivan, Mehmet Basoglu, Mahnaz Sadoughi, and Evgeni Krimer. We will share the memory of our time in a window-less room in ENS. I should also thank Mike for his numerous helps in writings.

I thank Hyunjee Yoon. She gave me the motivation and energy to power-through the last year of my graduate school.

Finally, I cannot express enough my gratitude to my parents, Gui Soo Jeong and Youn Sun Min. Their unconditional faith in me was the rock that I could hold on when in doubt of myself. I also thank my brother Minwoo Jeong for taking care of my family in my absence.

Min Kyu Jeong

December 2012, Austin, TX

Core-characteristic-aware Off-chip Memory Management in a Multicore System-on-Chip

Publication No. _____

Min Kyu Jeong, Ph.D.
The University of Texas at Austin, 2012

Supervisor: Mattan Erez

Future processors will integrate an increasing number of cores because the scaling of single-thread performance is limited and because smaller cores are more power efficient. Off-chip memory bandwidth that is shared between those many cores, however, scales slower than the transistor (and core) count does. As a result, in many future systems, off-chip bandwidth will become the bottleneck of heavy demand from multiple cores. Therefore, optimally managing the limited off-chip bandwidth is critical to achieving high performance and efficiency in future systems.

In this dissertation, I will develop techniques to optimize the shared use of limited off-chip memory bandwidth in chip-multiprocessors. I focus on issues that arise from the sharing and exploit the differences in memory access characteristics, such as locality, bandwidth requirement, and latency sensitivity, between the applications running in parallel and competing for the bandwidth.

First, I investigate how the shared use of memory by many cores can result in reduced spatial locality in memory accesses. I propose a technique that partitions the internal memory banks between cores in order to isolate their access streams and eliminate locality interference. The technique compensates for the reduced bank-level parallelism of each thread by employing memory sub-ranking to effectively increase the number of independent banks. For three different workload groups that consist of benchmarks with high spatial locality, low spatial locality, and mixes of the two, the average system efficiency improves by 10%, 7%, 9% for 2-rank systems, and 18%, 25%, 20% for 1-rank systems, respectively, over the baseline shared-bank system.

Next, I improve the performance of a heterogeneous system-on-chip (SoC) in which cores have distinct memory access characteristics. I develop a deadline-aware shared memory bandwidth management scheme for SoCs that have both CPU and GPU cores. I show that statically prioritizing the CPU can severely constrict GPU performance, and propose to dynamically adapt the priority of CPU and GPU memory requests based on the progress of GPU workload. The proposed dynamic bandwidth management scheme provides the target GPU performance while prioritizing CPU performance as much as possible, for any CPU-GPU workload combination with different complexities.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As processor arithmetic processing rates increase, corresponding increases in off-chip memory bandwidth are needed to realize the full potential improvements of application and system performance. The conventional rate of bandwidth scaling, however, leads to insufficient memory bandwidth for future processors with increasing number of cores. In this dissertation, I explore mechanisms to mitigate the bandwidth bottleneck of future systems with multicore processors. The mechanisms I develop exploit the differences in each core's memory access characteristics and requirements and improve the shared use of limited available bandwidth.

1.1 Multicore and Bandwidth Wall

With complexity and power consumption limiting improvements in single-thread performance, a recent design trend is to increase the number of cores to increase processor throughput. Such throughput-oriented parallel designs often require the off-chip memory bandwidth to scale roughly linearly with the core count, since each core may access private data and may not share any data with other cores. Another trend is to integrate heterogeneous

components of a system on a single chip (system-on-chip, or SoC) for faster and lower-overhead communication between the integrated components and for decreasing manufacturing cost. System components that were previously discrete with independent memory systems, now share the overall SoC bandwidth. Some SoC components, e.g. graphics processing unit cores, consume more bandwidth than general-purpose cores do. Such system-wide integration pushes the memory bandwidth requirement of a chip even higher than with typical general-purpose multi-core processors.

Scaling the off-chip memory bandwidth that is shared between the increasing number of cores is expensive. Package pin count and the off-chip signaling frequency can not scale linearly with the transistor count without paying excessive cost and power. The ITRS roadmap [5] predicts 5% annual increase in pin count, and over the DDR3 generation the signaling speed had increased about 20% per year. This combined bandwidth growth rate significantly lags behind the 100% increase in transistor count every 18 months as dictated by the Moore's law [63]. In many future systems, off-chip bandwidth will thus likely become the bottleneck of heavy bandwidth demand from multiple cores and the limiting factor for scaling the performance of a single chip. Therefore, intelligently managing the limited off-chip bandwidth is critical to achieving high performance and efficiency in future systems.

In addition to requiring a rapid increase in memory bandwidth, multicore systems differ in two key ways from their unicore predecessors with respect to their bandwidth usage. First, memory accesses come from indepen-

dent cores running in parallel, not from a single core running a single thread. Potentially independent access streams are interleaved at the shared off-chip memory interface. Locality in each access stream degrades as accesses from other streams conflict with the resources in use and come in between a stream's spatially adjacent accesses. This reduced locality results in lower performance and energy efficiency of the memory system, which has an increasingly large impact on the whole system throughput and power consumption.

The second key difference in bandwidth usage is that cores that access memory simultaneously can have very different memory access characteristics and requirements. For example, high-end SoCs now include powerful general purpose cores (CPU cores) and graphics processing cores (GPU cores). While both CPUs and GPUs are very demanding of the memory system, the CPU is latency sensitive and cannot tolerate long memory latencies without losing performance. The GPU, on the other hand, is designed to tolerate long latencies but requires consistent high bandwidth for periods of time to meet its real-time deadlines. Memory systems for conventional homogeneous processors are not designed to meet such diverse, often conflicting requirements of heterogeneous cores.

In summary, parallelism and heterogeneity in processors change the way the processors use off-chip bandwidth and will exacerbate the bandwidth wall problem. We need to take the parallelism and heterogeneity as a first class design consideration for memory systems for future systems.

1.2 Thesis Statement

By exploiting the different memory access characteristics and requirements of concurrent applications running on multiple cores, the interference between these applications can be managed better leading to memory systems for future multi-core processors that can achieve higher system performance and energy efficiency.

1.3 Contributions

In this dissertation, I develop mechanisms to optimize the shared use of limited off-chip memory bandwidth in chip-multiprocessors, and make the following contributions.

1. I propose a cost-effective technique that reduces the inter-thread interference at the shared memory system. Specifically, I show how the memory accesses of parallel threads interact with each other and affect the spatial locality and bank-level parallelism in a bandwidth-limited shared off-chip memory system. I propose a technique that partitions the internal memory banks between cores in order to isolate their access streams and to eliminate locality interference. The technique compensates for the reduced bank-level parallelism of each thread by employing memory sub-ranking which effectively increases the number of independent banks. This balanced approach is able to increase overall performance and power efficiency simultaneously.

2. I present a quantitative analysis of different memory address mapping strategies. I compare different mapping schemes, specified by the granularities at which the physical addresses are interleaved across DRAM structures, such as row, bank, rank, and channels. The trade-off options between the locality and parallelism, associated with the interleaving across different structures, are presented. I show how the locality and parallelism resulting from the mapping scheme affect the throughput and fairness in multicore systems and propose a mapping scheme that balances both for a representative system.

3. I propose a shared resource management approach for heterogeneous SoCs that exploits the different resource usage characteristics and application requirements. Specifically, I improve the off-chip memory bandwidth management in SoCs that have both CPU and GPU cores. I recognize that CPU and GPU cores have different latency sensitivity, average bandwidth consumption, and deadline requirements. I show that

   (a) Due to the nature of graphics workloads, bandwidth contention occurs at the beginning of each GPU frame, and

   (b) CPU performance can be improved in the presence of such contention by prioritizing its requests over requests of the GPU, as long as the frame deadlines are met.

   This simulation data, to the best of my knowledge, is the first publication available that shows how a GPU and a CPU interact at the shared

memory system with execution-driven models throttled by a feedback from the memory system.

4. I develop a GPU progress monitoring mechanism and enable dynamic adjustment of the priority policies discussed above in (3). I show that statically prioritizing the CPU can severely constrict GPU performance by allowing the CPU to monopolize shared bandwidth when a CPU workload requires high bandwidth for a long period of time. By monitoring the progress of the GPU, we can adapt the priority scheme dynamically and prioritize CPU requests only when it would not result in a missed deadline. Evaluation shows that such dynamic approach significantly improves GPU performance for workloads static prioritization fails, and provide the near-optimal QoS strategy for any combination of CPU-GPU workload demands. This dynamic GPU progress monitor can be used for other deadline-aware shared resource management schemes, or for dynamic voltage and frequency scaling of GPU.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 provides background for the modern DRAM system architecture and discusses prior work; Chapter 3 describes the locality interference issues in general purpose chip-multiprocessors; Chapter 4 discusses the bandwidth management for heterogeneous SoCs with cores of different requirements; and Chapter 5 concludes the dissertation and presents future research directions.

Chapter 2

Background and Related Work

In this chapter, I review background for the problem of the memory bandwidth bottleneck in future processors. I will describe memory system architecture, JEDEC DDR DRAM in particular, with an emphasis on how its structure makes memory system performance heavily dependent on access patterns. I then discuss fundamental memory controller design choices, such as physical-to-DRAM address mapping, row-buffer precharge policy, scheduling algorithms, and power-down management.

There has been a large body of prior work on memory systems and the bandwidth bottleneck problem. I will provide an overview of such related work in this chapter, and will discuss ones closely related to this dissertation in more detail within later chapters.

2.1 Memory System Architecture

DRAM-based main memory systems have hierarchical structures, from data arrays internal to a bank within a chip, up to the channels with many chips. The performance and energy efficiency of a DRAM memory system depend on how the access pattern exercises different parts of the hierarchy. I

will describe how each level of the hierarchy is designed, with its implications on performance and efficiency, starting with a single DRAM chip.

DRAM is a commodity memory device used for a variety of applications with varying requirements, from embedded systems to supercomputers. Since the production volume drives the cost, DRAM chips come in a few fixed configurations. The key DRAM chip parameters are: capacity (ex. 8Gb), data interface width (ex. x8), and signaling speed (ex. 1600MHz). For example, an 8Gb x8 DDR3-1600 chip, can provide a maximum bandwidth of 1.6GB/s and can store 1GB of data. Another important parameter for DDR DRAMs is burst length: one read command returns a burst length number of data beats, starting from the address provided. For DDR3, the burst length is eight. The example chip above returns 8 bits of data 8 times for each read command. Therefore, although it has 8-bit data interface, the minimum size of data accessible is 8B.

In general, one DRAM chip does not provide enough bandwidth and capacity for a general purpose system. For example, with a x8 chip, it takes 8 read commands to read a 64B cache line over 64 data cycles. To reduce the data transfer time, multiple chips are combined to form a desired interface width to the processor. With eight x8 chips forming a 64-bit wide interface, one read command can read a 64B cache line over 8 data cycles; the minimum burst for these chips. This set of chips that share control (address and command) signals are called a rank. Rank width determines the minimum access granularity of the memory system. Therefore, the rank width is tightly coupled with the

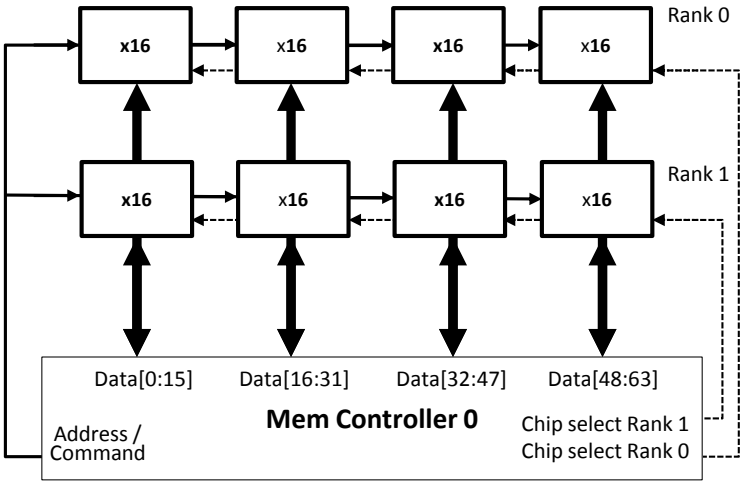


Figure 2.1: 64-bit DDR DRAM channel structure. The channel in this figure consists of two ranks, and each rank consists of four x16 chips.

access granularity of the processor, for example, the cache line size of its last-level cache. If the rank width were doubled to reduce the transfer time of 64B in half, the data returned per read request also doubles to 128B. Unless the last-level cache line size is increased, the additional pins and bandwidth would be wasted. If application spatial locality is low, additional data from bigger cache lines is not used and bandwidth and power to fetch them is wasted [75, 76].

To increase capacity, multiple ranks are put together. Ranks can share control signals and the data bus from the processor. Only one rank is controlled at any given time and is able to access the shared data bus. The set of ranks that share the control and data buses are called a channel. A significant downside of adding ranks is reduced signaling speed. Because the bus is shared, adding drops degrades signal integrity. Also, it takes extra time to switch

between ranks. To increase capacity without reducing signaling speed, buffers can be added at each rank to turn the bus into a daisy-chained point-to-point connection, e.g., FB-DIMM [49], buffer on board [3, 20, 79], registered DIMM, load-reduced DIMM [74]. For such designs, additional latency and power are consumed at these buffers.

Figure 2.1 shows a block diagram for a channel, with the structures described so far. For higher bandwidth without increasing the access granularity, multiple channels can be added. Since each channel has its own control and data bus, multiple channels can transfer data simultaneously. The cost to adding channels is extra pins from processors and additional control bandwidth from memory controller for independent control signals.

So far, we looked at the structures outside of a chip. There are hierarchical structures internal to a DRAM chip as well. A chip has multiple banks, four to sixteen for DDRx, so that multiple accesses to different banks can be overlapped. The cost to add additional banks is peripheral circuitry, such as address decoder and sense amplifiers, which is needed for each bank to handle an independent access.

A bank has data arrays and each storage cell can be specified by its row and column address. Figure 2.2 shows the internal structure of a chip with multiple banks, and a bank with a data array. To access a storage cell, first a row activation command to its row address reads an entire row into a row-buffer. Then, a column read command reads N columns, starting from the column address given, from the row-buffer out onto the data bus, where

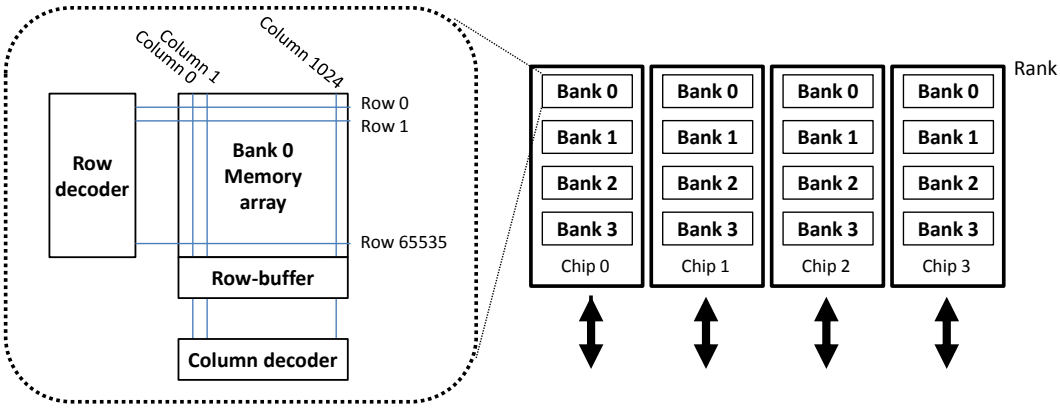


Figure 2.2: Memory bank structure. One rank from Figure 2.1 is shown in detail; each chip has four banks, each bank has 64K rows, and each row has 1K columns. The chip capacity is 64K (row) * 1K (columns) * 16 (width) = 1Gb.

N is the burst length. Once a row is open in the row-buffer, column read commands to the same row can be issued without re-opening the row. Such fast and energy efficient accesses are referred to as page-mode accesses.

2.2 Memory Controllers

A memory controller serves memory transactions from cores and peripherals to the memory system. The memory transactions from cores are simply specified as reads and writes to physical addresses. For each memory transaction, the memory controller's job is to

1. Translate a linear physical address into a multi-dimensional DRAM address.

2. According to the DRAM access protocol, issue necessary DRAM commands to the translated DRAM address.

3. Once the data returns from the DRAM, forward it to the requester.

Although it is simple for one transaction, there can be multiple memory transactions waiting to be serviced at the memory controller, especially for a memory system shared by multiple cores. Then the scheduling of these transactions becomes important. Because of the various DRAM command timing restrictions, the order the DRAM commands are scheduled can make a large difference on memory system performance and energy efficiency. This section describes some key design aspects of the memory controller that determine the scheduling.

2.2.1 Row-Buffer Precharge Policy

One of the basic scheduling policy decision is whether to precharge the row after it has been accessed. Open page polices leave rows open in anticipation of future row-hit accesses, and are therefore good for applications with high spatial locality. Close page policies precharge rows after the last access to that row (i.e. no pending requests to the row in the memory controller), so that a future access to a different row does not have to wait for the precharge. A close page policy is good for applications with low spatial locality. A hybrid policy, which leaves the row open and closes the row if it is not accessed for a while, have also been used [1, 29]. Several recent studies predict that with

the increasing number of cores, open-page policies will be less effective due to the interference between threads [72], or will be harmful to the system fairness due to possible starvation [35]. I will discuss these issues in more detail in Chapter 3.

2.2.2 Address Mapping

As described earlier, DRAM memory systems have a hierarchical structure. This hierarchy makes the DRAM address space multi-dimensional whereas the physical address space being mapped is linear. According to the address mapping, the memory controller translates a physical address into a channel, a rank, a bank, a row, and a column address. Typically, a bit-mask is used to extract each DRAM address field from physical address bits. For this scheme to work, the address ranges of the fields should be powers of two, which is true in most cases. Figure 2.3 shows two example bit-masks that a memory controller can use to translate a physical address. Figure 2.4 shows how the physical addresses (shown at a granularity of 64B block ID) are mapped to DRAM addresses by each mapping scheme of Figure 2.3. For the same physical address access pattern from the processor, the DRAM access behaviors with these two mappings differ greatly.

The position of each DRAM structure field in the map determines the granularity of interleaving across the structure. In Figure 2.3, both masks have a channel field at bit 6, thus every 64B is mapped to alternating channels. By interleaving at the finest granularity, most access patterns can utilize the pin

bandwidth of two channels and loads on the channels (i.e. pins) are more likely to be balanced. The bit-mask 0 maps 2^7 consecutive 64B blocks, across two channels, to the same rank, bank and row. The next 2^7 64B blocks are mapped to the next bank of the same rank. When the access pattern has spatial locality, this mapping will enable efficient page-mode accesses as the adjacent physical addresses are in the same row. On the other hand, the bit-mask 1 distributes 16 consecutive 64B blocks to all channels, ranks, and banks. In this case, parallelism across structures is prioritized. When the access pattern is random, this mapping will likely to result in more load-balanced banks than the bit-mask 0. Table 2.1 summarizes the benefits and costs involved with interleaving at each structure.

Table 2.1: DRAM structures and interleaving characteristics

Structure	Interleaving characteristics	
	Good	Bad
Channel	Fast switching Additional bandwidth from pins Adds channel-level parallelism	
Rank	Adds rank-level parallelism	Switching costs 1-2 cycles
Bank	Fast switching Adds bank-level parallelism	
Row		Slow switching Previous row should be closed No parallelism
Column	Fast switching Can reuse the row	No parallelism

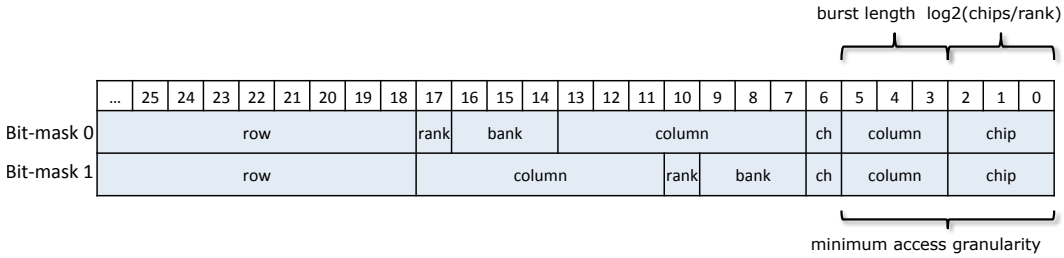
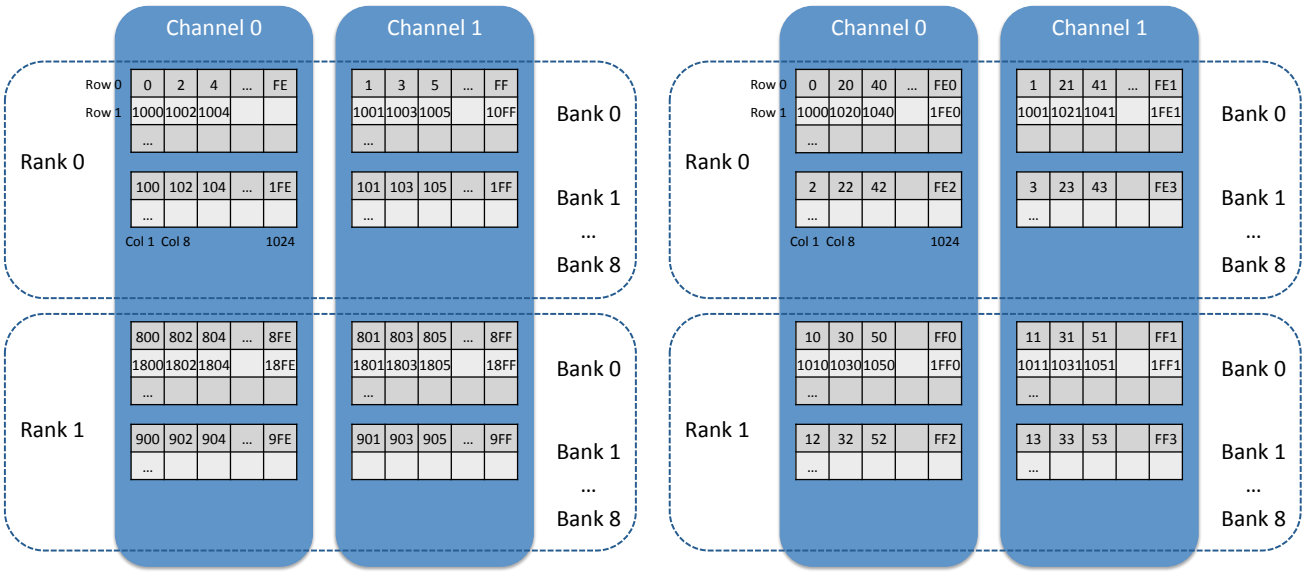


Figure 2.3: Two possible address mapping schemes for a memory system of: two channels, two ranks per channel, eight internal banks, x8 chips with a burst length of eight, 1024 columns per row. The numbers are the bit indices of a physical address.



(a) Physical address layout in DRAM by bit-map 0

(b) Physical address layout in DRAM by bit-map 1

Figure 2.4: Physical address (64B block number) layout in DRAM according to the address map shown in Figure 2.3.

2.2.3 Scheduling

FR-FCFS [62] is a widely used baseline scheduling algorithm that prioritizes transactions to open rows over transactions to still-closed rows. Transactions to the same row can be served more efficiently in page mode, even though they did not arrive at the memory controller together. Since the page mode accesses are lower latency and more energy efficient, most high-performance memory controllers implements the row-hit prioritization at some level.

2.2.4 Starvation Prevention

As requests are scheduled out-of-order and there are multiple cores issuing memory requests, starvation is possible when the scheduling algorithm favors certain access streams. The FR-FCFS algorithm could favor the row-buffer hit requests from certain cores for a long period of time, while a core with row-buffer misses keeps waiting. We need to have some mechanism to prevent a bank-conflict request from being starved indefinitely. We can use age-based promotion, so that a timed-out request is scheduled with top priority. Another approach in use is limiting the number of reads per activation, when there is a pending bank-conflict request. Or, we can limit the spatial locality and change the address mapping so that such long burst of reads to the same row do not occur [35].

2.2.5 Priority

Memory requests have different importance with respect to system performance. Demand read requests from general purpose cores, for example, are generally more performance critical than speculative prefetch requests. Delaying demand requests would most likely cost additional stalled cycles of originating cores. Likewise, requests from latency-insensitive GPU cores can be treated with different priorities.

The memory scheduler can take this priority information into account when making scheduling decisions. Lee et al. [44] shows that the priority of prefetches relative to demand requests should be dependent on how useful the prefetches are. Kaseridis et al. [35] proposes that among the demand requests, the memory level parallelism (MLP) of the issuing core could be used to further classifying the priority of the requests. Cores with high MLP are less memory sensitive than the ones with low MLP. Also, the prefetch requests are prioritized with, the prefetch distances are used to estimate the latency sensitivity.

2.2.6 DRAM Power Management

The memory system is consuming an increasingly large portion of total system power, already 25-40% for large-scale datacenters [21, 23, 46]. DDR DRAMs have power-down modes chips can be in when the they are idle. Memory controllers can save significant amount of DRAM background power by utilizing the DRAM power-down states.

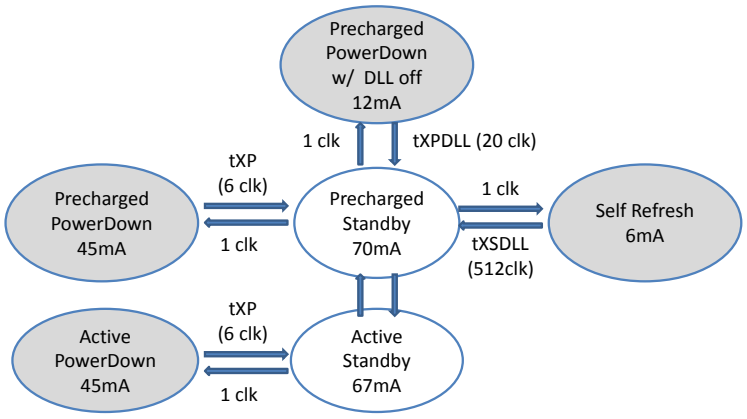


Figure 2.5: DDR 3 power-down state transition diagram. Entering a power-down mode is instant, but exiting could take a while for a deep power-down or self-refresh mode. The less power a mode consumes, the longer it takes to exit.

Entering and exiting a power-down mode takes time. Figure 2.5 shows DRAM power-down states and the time it takes to transition between them. Since there are multiple power-down modes with varying degree of power saving and cost of transitions, several strategies have been proposed for controlling DRAM power. One example is powering-down only when the rank has been idle for a period of time [42]. Many studies suggest, however, that entering a power-down mode immediately when there are no pending requests to the rank works best [26, 80].

2.3 Related Work

2.3.1 Bandwidth Wall

Several works have identified the bandwidth wall problem and studied the design space of future CMPs limited by the off-chip memory bandwidth.

Huh et al. [25] showed that due to the bandwidth scaling gap, fewer out-of-order processing cores will have higher throughput than more in-order cores on future CMPs. Also, they predicted that the increasing transistor/signal pin ratio will necessitate larger on-chip caches per core, prohibiting linear scaling of core count. More recently, Rogers et al. [63] also studied the scaling of larger-scale CMP processors with an analytical model, and reached a similar conclusion regarding the die area budget for cores and cache. Their analysis shows the future processors can scale the number of cores only by 3x, in contrast to 16x, over four technology generations under the current bandwidth envelope. The effectiveness of several bandwidth-saving techniques, including eDRAM cache, link/cache compression, and 3D stacked caches, were evaluated with respect to mitigating the bandwidth bottleneck. These approaches either try to reduce the bandwidth consumption or increase the effective bandwidth, whereas the techniques proposed in this dissertation maximizes processor performance and energy efficiency under the given budget.

Kaseridis et al. [34] proposed a dynamic process migration among processors in a multi-socket system. Since each socket has its own memory channels, load-balancing the bandwidth requirements of processes running in each processor can improve the overall system throughput. They used resource profilers to dynamically measure the contention at each memory channel and migrated processes away from over-committed processors to more available ones.

2.3.2 Memory Controller Scheduling Policies

A number of memory controller scheduling policies have recently been proposed that aim at improving fairness and system throughput in shared-memory CMP systems. Computer-network-based fair queuing algorithms using virtual start and finish times are proposed in [58] and [61], respectively, to provide QoS for each thread. STFM [54] makes scheduling decisions based on the stall time of each thread to achieve fairness. A parallelism-aware batch scheduling method is introduced in [55] to achieve a balance between fairness and throughput. It groups requests into batches based on their arrival time and process earlier batches first to provide fairness. ATLAS [37] prioritizes threads that attained the least service from memory controllers to maximize system throughput at the cost of fairness. TCM [38] divides threads into two separate clusters and employs a different memory request scheduling policies to each cluster. Minimalist open page [35], solves a fairness issue of FR-FCFS with a different address mapping scheme that limits the amount of spatial locality exploitable by page mode accesses. Then it proposes a priority scheme based on the memory-level parallelism of each core, and the prefetch distance of the prefetchers.

This previous work addresses the same problem of inter-thread interference in a shared memory system, but focuses on improving the *scheduling* of shared resources to threads waiting to be serviced. By considering their memory access behavior and system fairness, a better scheduling decision can be made at the memory controller. The techniques presented in this dissertation

are orthogonal to the existing scheduling policy work, as they focus on improving efficiency by preserving row-buffer locality, and on balancing parallelism with access latency. Bank partitioning can benefit from better scheduling, and better scheduling policies can benefit from improved row-buffer spatial locality. The QoS scheme is for heterogeneous CPU and GPU cores whereas the previous work targets homogeneous general-purpose CMPs.

Chapter 3

Preserving Spatial Locality in CMP systems

Modern main memory systems rely on spatial locality to provide high bandwidth while minimizing memory device power and cost. Exploiting spatial locality improves bandwidth and efficiency in four major ways: (1) DRAM addresses can be split into row and column addresses which are sent to the memory devices separately to save address pins; (2) access granularity can be large to amortize control and redundancy overhead; (3) *page mode* is enabled, in which an entire memory row is saved in the row-buffer so that subsequent requests with the same row address need only send column addresses; and (4) such *row-buffer hit* requests consume significantly less energy and have lower latency because the main data array is not accessed.

Many applications present memory access patterns with high spatial locality and benefit from the efficiency and performance enabled by page mode accesses. However, with the increasing number of cores on a chip, memory access streams have lower spatial locality because access streams of independent threads may be interleaved at the memory controller [10, 72]. When two independent memory access streams with high spatial locality access two different rows of the same bank, they suffer in a long period of bank conflicts.

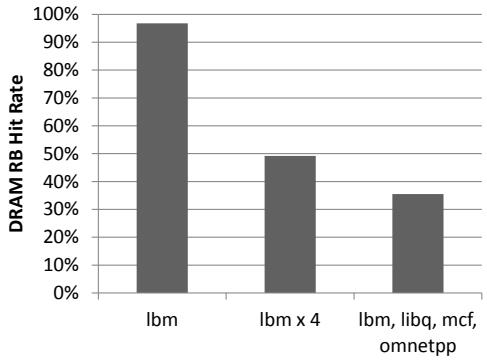


Figure 3.1: DRAM row-buffer hit rate of `lbm` when run alone, with 4 instances, and when run with other applications. FR-FCFS scheduling is used for higher hit rate.

The bank has to alternate between the streams, and every time it switches the row it buffers, additional energy and delay to precharge and activate the rows are wasted. Figure 3.1 shows, for example, the impact of access interleaving on the spatial locality of the SPEC CPU2006 benchmark `lbm`. The DRAM row-buffer hit rate of one `lbm` instance is shown for three different workload mixes. When `lbm` runs alone, 98% of the accesses hit in the row-buffer. When 4 instances of `lbm` run together, however, the hit rate drops to 50%. The spatial locality of an interleaved stream can drop even further depending on the application mix, because some applications are more intrusive than others. The SPEC CPU2006 benchmark `mcf`, for example, is memory intensive but has very poor spatial locality, and thus severely interferes with the memory accesses of other applications. In a multi-programmed workload containing `mcf`, the row-buffer hit rate of `lbm` drops further to 35%. Loss of spatial locality due to inter-thread interference increases energy consumption because of

additional row activations, increases the average access latency, and typically lowers the data throughput of the DRAM system.

Out-of-order memory schedulers reorder memory operations to improve performance and can potentially recover some of the lost spatial locality. However, reclaiming lost locality is not the primary design consideration for these schedulers. Instead, they are designed either to simply maximize the memory throughput for a given interleaved stream (e.g., FR-FCFS [62]), or to maximize some notion of fairness and application throughput (e.g., ATLAS [37] and TCM [38]). In other words, improved scheduling can somewhat decrease the detrimental impact of interleaving but does not address the issue at its root. Furthermore, the effectiveness of memory access scheduling in recovering locality is constrained by the limited scheduling buffer size and the (often large) arrival interval of requests from a single stream. As the number of cores per chip grows, reordering will become less effective because the buffer size does not scale and contention for the shared on-chip network can increase the request arrival interval of each thread. With an increased arrival interval, subsequent spatially adjacent requests have a higher chance of missing the scheduling window, and the row is precharged prematurely.

In this chapter, I will present a fundamentally different approach that addresses locality interference at its root cause. Rather than recovering limited locality with scheduling heuristics, I propose to reduce (and possibly eliminate) row-buffer interference by restricting interfering applications to non-overlapping sets of memory banks [32]. In some cases, however, restricting

the number of banks available to an application can significantly degrade its performance as fewer banks are available to support concurrent overlapping memory requests. This is of particular concern because the total number of banks available to a processor is growing more slowly than the total number of threads it can concurrently execute. Therefore, I combine *bank partitioning* with memory sub-ranking [10, 11, 18, 73, 78], which effectively increases the number of independent banks. Together, bank partitioning and memory sub-ranking offer two separate tuning knobs that can be used to balance the conflicting demands for row-buffer locality and bank parallelism. I will show that this powerful combination, unlike memory sub-ranking or bank partitioning alone, is able to simultaneously increase overall performance and significantly reduce memory power consumption.

The rest of this chapter is organized as follows: Section 3.1 discusses how the DRAM address mapping affects locality interference and describes our bank partitioning mechanism. Section 3.2 describes how bank partitioning reduces the available DRAM bank-level parallelism, and investigates how sub-ranking can recover the lost parallelism. Section 3.3 reviews related work. Section 3.4 shows our evaluation and discusses results. I discuss the scalability of the bank partitioning and exploiting spatial locality altogether in future many-core systems in Section 3.5. Different address mapping strategies are compared in Section 3.6 with regard to spatial locality exploited and their impact in performance. Section 3.7 concludes the chapter.

3.1 Bank Partitioned Address Mapping

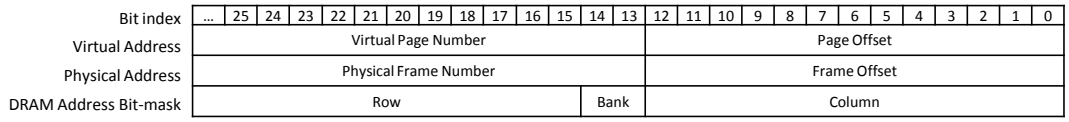
Row-buffer locality interference in multicore processors results from the mechanisms used to map the addresses of threads onto physical memory components. This mapping is determined by the combination of the OS and the memory controller, which respectively translates virtual to physical address, and physical to DRAM address. Current mapping methods evolved from uniprocessors, which do not finely interleave independent address streams. These methods are sub-optimal for chip multiprocessors where memory requests of applications can interfere with one another. In this section, I will first describe the mechanism commonly in use today and then discuss our proposed mapping mechanism that simultaneously improves system throughput and energy efficiency.

3.1.1 Current Shared-bank Address Mapping

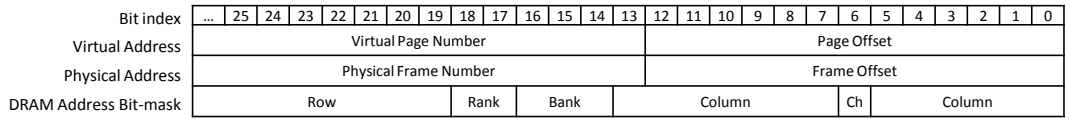
In traditional single-threaded uniprocessor systems with no rank-to-rank switching penalty, memory system performance increases monotonically with the number of available memory banks. This performance increase is due to the fact that bank-level parallelism can be used to pipeline memory requests and hide the latency of accessing the inherently slow memory arrays. To maximize such overlap, current memory architectures interleave addresses among banks and ranks at fine granularity in an attempt to distribute the accesses of each thread across banks as evenly as possible. However, because row-buffer locality can be exploited to improve efficiency and performance, it

is common to map adjacent physical addresses to the same DRAM row so that spatially close future accesses hit in the row-buffer. Therefore, the granularity of bank interleaving in this case is equal to the DRAM row size. Real-time systems often use cache-line sized bank interleaving with closed-page policy for predictable latency. However, a strict closed-page policy usually has lower performance and energy efficiency than a open-page policy [14, 80], thus not considered in general-purpose systems.

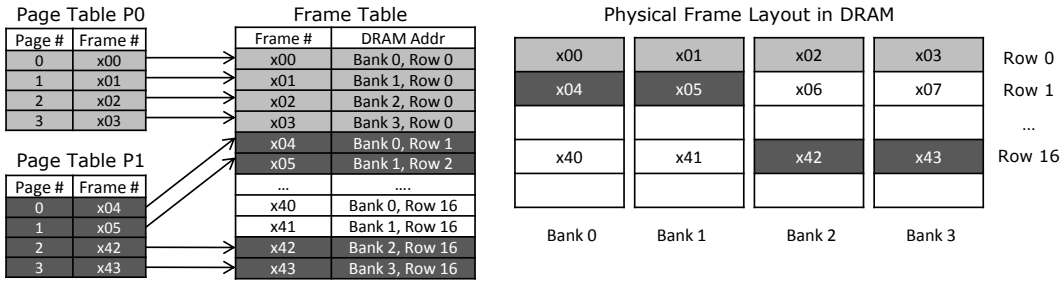
Figure 3.2 gives an example of the typical row-interleaved mapping from a virtual address, through a physical address, to a main memory cell. The mapping given in Figure 3.2(a) is for a simple memory system (1 channel, 1 rank, and 4 banks) and is chosen to simplify the following discussion. The mapping in Figure 3.2(b) is given for a more realistic DDR3 memory system with 2 channels, 4 ranks, and 8 banks. When two threads concurrently access memory, both threads may access the same memory bank because the address space of each thread maps evenly across all banks. Figure 3.2(c) depicts the full address translation process and shows how two threads with the simple mapping (Figure 3.2(a)) can map to the same bank simultaneously. When thread P0 and P1 accesses their virtual pages 2 and 3, they conflict in banks 2 and 3. Bank conflicts, such as this one, thrash the row-buffers and can significantly reduce the row-buffer hit rate. The likelihood of this type of interference grows with the number of cores as more access streams are interleaved.



(a) Virtual to physical to DRAM address mapping scheme for a simple memory system.



(b) Generalized Virtual to physical to DRAM address mapping scheme.

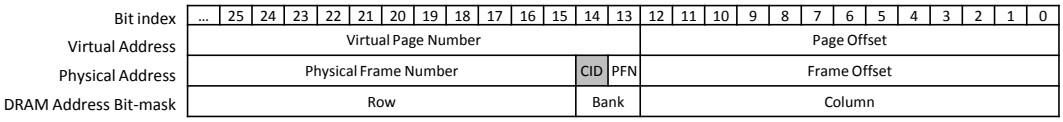


(c) The layout of virtual pages allocated to each process in the DRAM banks according to the map (a).

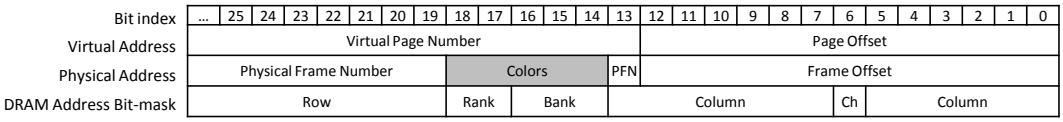
Figure 3.2: Conventional address mapping scheme

3.1.2 Bank Partitioned Address Mapping

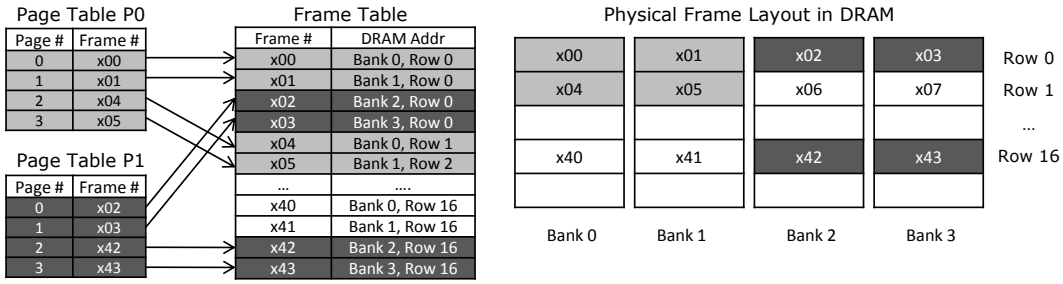
If independent threads are not physically mapped to the same bank, inter-thread access interference cannot occur. I will refer to this interference avoidance mechanism as *bank partitioning* as proposed by Mi et al. [50]. Figure 3.3(a) gives an example address mapping scheme for a simple system which utilizes bank partitioning. In this example, bit 14 of the physical address denotes the core ID. Core 0 is allocated physical frames $\{0, 1, 4, 5, 8, 9, ...\}$, and core 1 is allocated frames $\{2, 3, 6, 7, 10, 11, ...\}$. This mapping partitions the banks into two sets such that processes running on different cores are con-



(a) Virtual to physical to DRAM address mapping scheme for a simple memory system.



(b) Generalized Virtual to physical to DRAM address mapping scheme.



(c) The layout of virtual pages allocated to each process in the DRAM banks according to the map (a).

Figure 3.3: Address mapping scheme that partitions banks between processes.

strained to disjoint banks, thus avoiding row-buffer interference. Figure 3.3(c) shows the corresponding change in virtual page layout in DRAM and illustrates how bank interference is eliminated. Virtual pages in P0 and P1 are mapped to only banks $\{0, 1\}$ and $\{2, 3\}$, respectively. Therefore, they do not share row-buffers and cannot interfere with one another. Figure 3.4 gives an example of memory behavior with and without bank partitioning. P0 and P1 concurrently access their virtual page #2 in an interleaved manner; bank partitioning eliminates the precharge commands and the second activate command for P0. This improves both throughput and power efficiency relative to the memory behavior using conventional frame allocation.

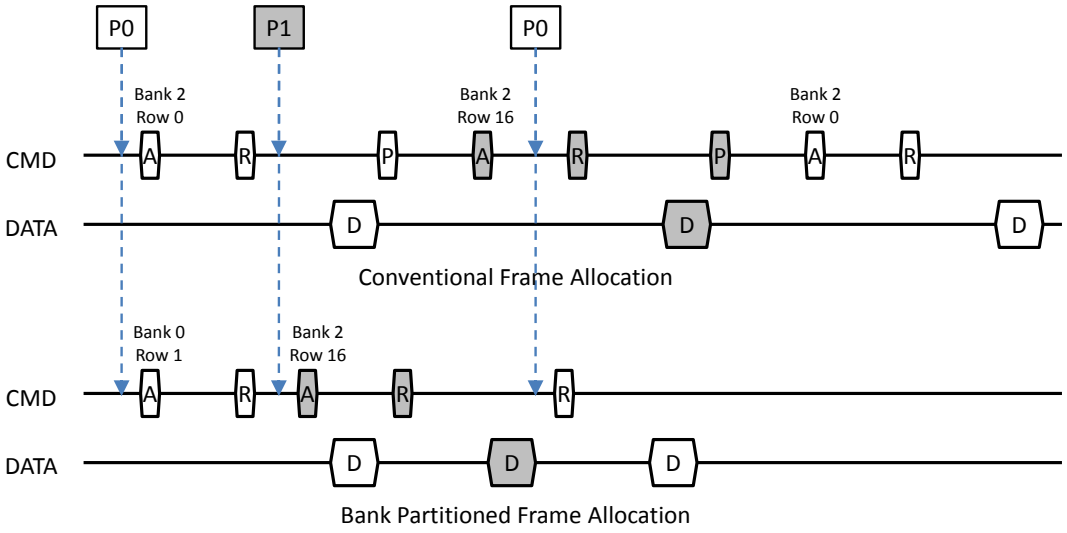


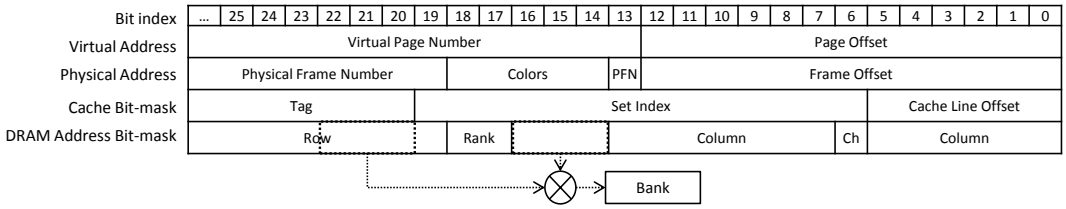
Figure 3.4: Benefits of bank partitioning. The upper diagram corresponds to Figure 3.2(c), and the lower diagram corresponds to Figure 3.3(c). A, R, P, and D stand for activate, read, precharge, and data, respectively. The row-buffer is initially empty.

Figure 3.3(b) shows a generalized bank partitioning address mapping scheme for a typical DDR3 memory system with 2 channels, 4 ranks, and 8 banks. Different *colors* represent independent groups of banks. Given a number of colors, there is a wide decision space of allocating colors to processes. We can imagine an algorithm that allocates colors depending on the varying applications' need for bank-level parallelism. However, this study uses a static partition which assigns an equal number of colors to each core. Such static partitioning does not require profiling of workloads and is robust to dynamic workload changes. I found that there is a diminishing return in increasing the number of banks that a modern out-of-order core can exploit from more bank-level parallelism. For realistic modern memory system configurations which

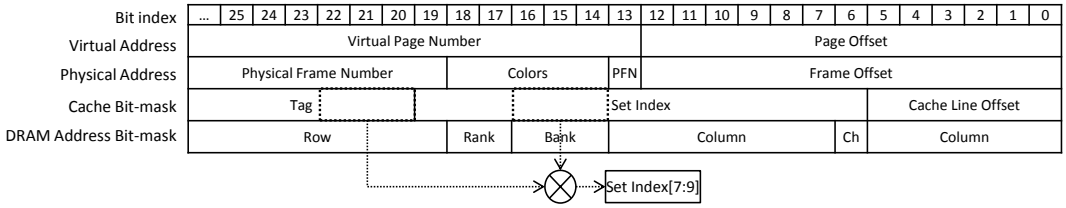
have internal banks and consist of multiple channels and ranks, sub-ranking can compensate for reduced per-thread bank count. More details on bank-level parallelism will be discussed in next section.

The mapping between physical and DRAM addresses stays the same, allowing the address decoding logic in the memory controller to remain unchanged. Bank isolation is entirely provided by the virtual to physical address mapping routine of the operating system. Therefore, when a core runs out of physical frames in colors assigned to it, the OS can allocate different colored frames at the cost of reduced isolation.

XOR-permuted bank indexing is a common memory optimization which reduces bank conflicts for cache-induced access patterns [77]. Without any permutation, the set index of a physical address contains the bank index sent to DRAM. This maps all cache lines in a set to the same memory bank, and guarantees that conflict-miss induced traffic and write-backs will result in bank conflicts. XOR-based permutation takes a subset of the cache tag and hashes it with the original bank index such that lines in the same cache set are mapped across all banks. XOR-permuted bank indexing nullifies color-based bank partitioning and re-scatters the frames of a thread across all banks. I use a technique proposed by Mi et al. [50] to maintain bank partitioning. Instead of permuting the memory bank index, we permute the cache set index. In this way, a single color still maps to the desired set of banks. At the same time, cache lines within a set are mapped across multiple banks and caches sets are not partitioned. Figure 3.5 demonstrates the virtual-to-physical



(a) XOR Permutation-based DRAM bank indexing scheme



(b) XOR Permutation-based cache set indexing scheme

Figure 3.5: With the XOR bank permutation scheme as in (a), frames of one color are scattered to all banks. An alternative scheme (b) hashes the cache set index instead, preserving a 1:1 mapping between colors and DRAM banks.

address mapping scheme used by XOR-permuted bank indexing, as well as the permuted set indexing used in this study.

3.2 Bank-Level Parallelism

Bank partitioning reduces the number of banks available to each thread. This section examines the impact that reduced bank-level parallelism has on thread performance, and investigates the ability of memory sub-ranking to add more banks to the system.

3.2.1 Bank-level Parallelism

The exploitation of bank-level parallelism is essential to modern DRAM system performance. The memory system can hide the long latencies of row-buffer misses by overlapping accesses to many memory banks, ranks, and channels.

In modern out-of-order processor systems, a main memory access which reaches the head of the reorder buffer (ROB) will cause a structural stall. Given the disparity of off-chip memory speeds, most main memory accesses reach the head of the ROB, negatively impacting performance [33]. Furthermore, memory requests that miss in the open row-buffer require additional precharge and activate latencies which further stall the waiting processor. If a thread with low spatial locality generates multiple memory requests to different banks, they can be accessed in parallel by the memory system. This hides the latency of all but the first memory request, and can dramatically increase performance.

A thread, especially one with low spatial locality, can benefit from many banks up to the point where its latency is completely hidden. With DDR3-1600, a row-buffer miss takes about 33 DRAM clock cycles before it returns data (in the following 4 DRAM clock cycles). Therefore, to completely hide the latency of subsequent requests, a thread needs to have 8 banks working in

parallel [1] [2]. In general, as more requests are overlapped, the CPU will stall for less time and performance will increase.

While bank-level parallelism undoubtedly improves performance, its benefits have a limit. After the available parallelism can hide memory latencies, there is no further benefit to increasing the number of banks. Figure 3.6 shows applications' sensitivity to the number of banks in the memory system. Most applications give peak performance at 8 or 16 banks. After this point, applications show no further improvement or even perform worse, due to the 2 cycle rank-to-rank switching delay. `lbm` is an outlier that benefits beyond 16 banks. `lbm` shows a higher row-buffer hit-rate as the number of banks increases. It accesses multiple address streams concurrently, and additional banks keep more row-buffers open and can fully exploit each stream's spatial locality.

3.2.2 Sub-ranking

The most straightforward way to retain sufficient bank-level parallelism with bank partitioning is to increase the total number of banks in the memory system. The total number of banks in the system is equal to the product of the number of channels, the number of ranks per channel, and the number

[1] tFAW limits the number of activate commands to 6 per row-buffer miss latency, due to power limitations. Also, tRRD has increased so that activates should be separated by more than 4-cycle data burst, causing idle cycles on the data bus in the case that all requests miss in the row-buffer.

[2] Since a row-buffer miss takes a constant latency, more bank-level parallelism is needed with faster interfaces such as DDR3-1866 and 2133.

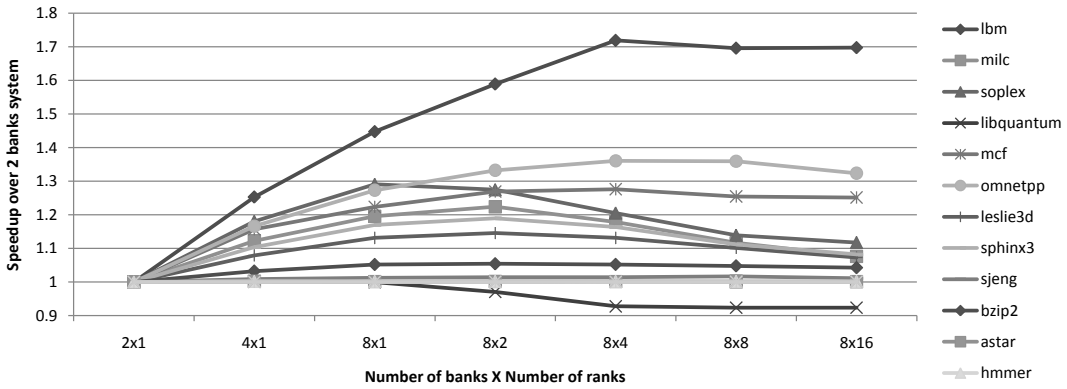


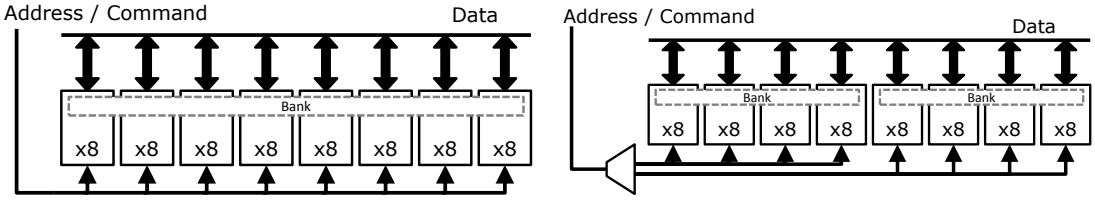
Figure 3.6: Normalized application execution time with varying number of banks.

of internal banks per chip. Unfortunately, increasing the number of available banks is more expensive than simply adding extra DIMMs.

There are a number of factors which complicate the addition of more memory banks to a system. First, and most fundamentally, paying for additional DRAM chips and power just to increase the number of banks is not a cost-efficient solution for system designers. Also, the number of channels is limited by the available CPU package pins. Furthermore, when fine-grained channel interleaving is used to maximize data link bandwidth, the contribution of the channel count to the effective number of banks is limited. In addition, the number of ranks per channel is constrained by signal integrity limitations. Point-to-point topologies as used in FB-DIMMs [49] can overcome this problem, but come at a cost of additional latency and power consumption. Finally, adding internal banks to each DRAM chip requires expensive circuitry changes. Considering the extreme competition and low margin of the DRAM

market, more internal banks would be difficult to achieve in a cost effective manner. For these reasons, with continued device scaling and the recent trend of increasing core count, future systems are expected to have higher core to bank ratio than current systems.

I propose an alternative approach to increase the bank-level parallelism without adding expensive resources. Custom DIMMs are widely used in many high-performance systems already [18, 27]. I employ DRAM sub-ranking [10, 73, 78] to effectively increase the number of banks available to each thread. DRAM sub-ranking breaks a wide (64-bit) rank into narrower (8-bit, 16-bit, or 32-bit) sub-ranks, allowing individual control of each sub-rank. Figure 3.7 shows an example of a DRAM rank with and without sub-ranking (with two sub-ranks). In the conventional memory system, all banks within a rank work in unison and hold the same row. In the 32-bit sub-ranked system, two groups of four banks are independent and can hold different rows. By controlling the sub-ranks independently, multiple long-latency misses can overlap, effectively increasing the available bank-level parallelism. However, since fewer devices



(a) Conventional DRAM rank. (b) Sub-ranked DRAM rank.

Figure 3.7: Comparison of a conventional memory system, and a sub-ranked memory system similar to MC-DIMM [10].

are involved in each request, sub-ranked DRAM takes more reads (and thus a longer latency) to complete the same size of request.

Although narrow 16-bit and 8-bit sub-ranks can provide abundant bank-level parallelism, additional access latencies due to sub-ranking become prohibitive. Given a 4GHz processor, 16-bit and 8-bit sub-ranks take 40 and 80 additional CPU cycles, respectively, to access main memory. In addition, narrow sub-ranks put greater pressure on the address bus, which could impact performance or necessitate expensive changes to DRAM. Accordingly, 32-bit sub-ranks are used to balance bank-level parallelism and access latency, and avoid the need for extra address bus bandwidth.

3.3 Related Work

3.3.1 Memory Partitioning

Kurian et al. [41] proposed a data placement technique which eliminates bank-conflict for applications accessing multiple arrays concurrently. By placing the arrays to disjoint partitions of external memory banks, inter-array bank-conflicts can be eliminated and memory utilization is improved. Recently, Mi et al. [50] proposed to partition DRAM banks to reduce interferences between threads running on a multi-core system. They use page coloring and XOR set-index hashing to allocate disjoint set of banks to different threads. They develop a cost model and search the entire color assignment space for each workload off-line. While this approach can approximate a near-optimal color assignment for a specific memory system and workload mix, the cost of

this search is prohibitive for dynamically changing workload mixes. Also, Mi et al. do not consider the impact of reduced bank-level parallelism; rather, they assume a very large number of banks, exceeding the practical limit of modern DRAM systems. The number of banks can be increased by adding more ranks to a system, but rank-to-rank switching penalties can limit performance as I show in Figure 3.6. I propose a sub-ranking-based alternative which is both cost-effective as well as power efficient, and provide in-depth analysis of the interaction between locality and parallelism. Recently, Liu et al. [47] implemented the bank-partitioning in a Linux kernel 2.6.32.15. They developed a small program that can figure out the bank address hashing function used in the host memory controller. They evaluted the bank-partitioning frame allocation on a real system and showed it can benefit both the multi-programmed and multi-threaded workloads.

Another existing approach by Muralidhara et al. [53] partitions memory channels instead of banks. Their partitioning method is based on the runtime profiling of application memory access patterns, combined with a memory scheduling policy to improve system throughput. However, they do not account for or analyze the impact of reduced bank-level parallelism. Also, their method interferes with fine-grained DRAM channel interleaving and limits the peak memory bandwidth of individual applications. To the best of our knowledge, our work is the first to preserve both spatial locality and bank-level parallelism.

3.3.2 DRAM-architecture-aware Frame Allocation

DRAM-architecture-aware frame allocation techniques have been proposed to reduce the main memory power consumption. Lebeck et al. [43] proposed a power-aware physical frame allocation policy for the operating system that exhausts free physical frames of one rank before starting to allocate for the next rank so that DRAM chips in the unused ranks can be put into low power modes. Huang et al. [24] extended the basic power-aware frame allocation policy to a single core multiprogrammed environment. They allocated physical frames shared among processes, such as shared libraries, together in the same ranks. Both methods are targeted at reducing DRAM background power whereas, in our approach, frame allocation is done to preserve spatial locality and balance bank-level parallelism.

3.3.3 Sub-ranking

Recently, several schemes to reduce the active power of DRAM have been proposed. Sub-ranking is a technique that breaks the conventional wide (64-bit) rank into narrower (8-bit, 16-bit or 32-bit) sub-ranks. As each access involves fewer DRAM chips, it reduces the power consumption significantly.

Ware and Hampel [73] propose the Threaded Memory Module, in which the memory controller itself controls individual sub-ranks with chip-enable signals. They analytically present the potential performance improvement due to the effective increase in the number of memory banks. Zheng et al. [78] propose Mini-Rank memory. They keep the memory controller module interface

the same, and access sub-ranks using a module-side controller. They focused on savings in activate power from sub-ranking. Ahn et al. [10, 11] propose MC-DIMM. Their work is similar to the Threaded Memory Module, except demultiplexing and the repetitive issue of sub-rank commands are handled on the module side. Also, Ahn et al. empirically evaluate MC-DIMM on multi-threaded applications, with full-system power evaluation. Their target is a heavily threaded system which can naturally tolerate the additional latency from narrow sub-ranks. Unlike the previous works, I focus on the additional bank-level parallelism that sub-ranking can provide in the context of latency sensitive, general purpose out-of-order processor systems.

3.3.4 Row-buffer Locality

Stuecheli et al. [70] proposed a proactive write scheduling to improve the row-buffer locality of write requests. In cache-based systems, a write to the memory occurs when a last-level cache line is evicted, not when a processor stores a new value. A write stream consists of the requests that happen to be evicted around the same time-frame, independent to the access pattern of applications, thus lacks spatial locality. They propose to actively cleaning the dirty cache lines in the last-level cache that maps to the same row. By increasing the write bursts, DRAM utilization improves and read-write turnaround is reduced.

3.4 Evaluation

In this section, I present the evaluation of the bank-partitioning and sub-ranking mechanisms. The evaluation methodology, such as simulators, power models, workloads and metrics are described first. The experimental results that show the impact of the proposed techniques on system performance and energy are presented next.

3.4.1 Methodology

The proposed mechanisms are evaluated using cycle-based system simulations with a cycle-level x86 out-of-order processor simulator, Zesto [48], and the DrSim DRAM simulator [31]. Zesto runs user space applications bare-metal and emulates system calls. I extended the physical frame allocation logic of Zesto to implement bank partitioning among cores as presented in Section 3.1.2. The DrSim DRAM simulator supports sub-ranked memory systems. It models memory controllers and DRAM modules faithfully, simulating the buffering of requests, scheduling of DRAM commands, contention on shared resources (such as address/command and data buses), and all latency and timing constraints of DDR3 DRAM. It also supports XOR interleaving of bank and sub-rank index to minimize conflicts [77], which is used for the baseline shared-bank configuration and replaced with cache-set index permutation described in 3.1.2 for bank-partitioned configurations.

Table 3.1: Simulated system parameters

Processor	8-core, 4GHz x86 out-of-order, 4-wide
L1 I-caches	32KiB private, 4-way, 64B line size, 2-cycle
L1 D-caches	32KiB private, 8-way, 64B line size, 2-cycle
L2 caches	256KiB private for instruction and data, 8-way 64B line size, 6-cycle
L3 caches	8MiB shared, 16-way, 64B line size, 20-cycle
Memory controller	FR-FCFS scheduling, open row policy 64 entries read queue, 64 entries write queue
Main memory	2 channels, 2 ranks / channel, 8 banks / rank, 8 x8 DDR3-1600 chips / rank All parameters from the Micron datasheet [52]

System Configuration

Table 4.1 summarizes the system parameters of our simulated systems. I simulate the following 4 configurations to evaluate our proposed mechanism.

1. `shared`: Baseline shared-bank system.
2. `bpart`: Banks partitioned among cores. Each core receives consecutive banks from one rank.
3. `sr`: Ranks split into two independent sub-ranks.
4. `bpart+sr`: Bank partitioning with sub-ranking used together.

Power models

I estimate DRAM power consumption using a power model developed by the Micron Corporation [2]. For processor power, I use the IPC-based estimation presented in [10]: the maximum TDP of a 3.9GHz 8-core AMD bulldozer processors is reported as 125W; half of the maximum power is assumed to be static (including leakage) and the other half is dynamic power

that is proportional to IPC. Our study focuses on main memory and, as such, the proposed mechanisms have a minimal impact on the core power.

The memory controller puts any idle rank into fast a power-down mode to save power immediately when there is no pending requests to the rank, as suggested by previous work [26, 80].

Workloads

I use multi-programmed workloads consisting of benchmarks from the SPEC CPU2006 suite [68] for evaluation. Our evaluation is limited to multi-programmed workloads due to the limitation of our infrastructure. The principle of placing memory regions that interfere in access scheduling in different banks to better exploit spatial locality, however, applies to both multi-programming and multi-threading. Identifying conflicting regions can be more challenging with multi-threading, yet doable (e.g., parallel domain decomposition can indicate bank partitions; and thread-private data is often significant).

To reduce simulation time, I use SimPoint [65] and determine a representative 200 million instruction region from each application. Memory statistics from identified region are used to guide the mix of applications in our multi-programmed workloads. Table 3.2 summarizes the characteristics of each application. Some benchmarks are not included in our evaluation due to limitations of Zesto.

Two key application characteristics which are pertinent to our evaluation are the memory access intensity and the row-buffer spatial locality.

The memory access intensity, represented by last-level cache misses-per-kilo-instructions(LLC MPKI), is an indicator of how much an application is affected by the memory system performance. I focus most our evaluation on workloads that include memory intensive applications.

The spatial locality of application, represented by the row-buffer hit rate, is another key characteristic. Applications with high row-buffer hit rates suffer the most from row-buffer locality interference; conversely, they stand to benefit the most from bank partitioning. On the other hand, applications with low row-buffer hit rates rely on bank-level parallelism, and may be adversely affected by any reduction in the number of available banks. Among the memory-intensive benchmarks, `mcf` and `omnetpp` have low spatial locality. `libquantum`, `lbm`, `milc`, `soplex` and `leslie3d` all demonstrate high spatial locality in the absence of interference.

Table 3.3 shows the mix of benchmarks chosen for our multi-programmed workloads. Unless otherwise noted, all benchmark mixes contain programs with high memory intensity. Group `HIGH` consists of workloads with high spatial locality. Group `MIX` is a mixed workload of programs with both high and low spatial locality. Group `LOW` consists of workloads with low spatial locality. Finally, group `LOW_BW` is composed of workloads that contain non-memory-intensive benchmarks to see the effect of our mechanisms on compute-bound workloads. Each mix in the table contains one, two, or four benchmarks. These benchmark mixes are replicated to have eight benchmarks to run on each of the eight cores in our simulated system.

Table 3.2: Benchmarks statistics when running on the baseline. IPC: Instructions per cycle, LLC MPKI: Last level-cache misses per 1000 instructions, RB Hit Rate: Row-buffer hit rate, and Mem BW: Memory bandwidth used in MiB/S.

Benchmark	IPC	LLC MPKI	RB Hit Rate	Mem BW
lbm	0.71	12.80	97%	3394.56
milc	0.69	16.15	82%	3576.32
soplex	0.53	20.88	90%	3246.08
libquantum	0.53	15.64	98%	2967.04
mcf	0.54	16.15	10%	2258.94
omnetpp	0.69	9.51	49%	2136.83
leslie3d	0.73	8.04	89%	1942.53
sphinx3	1.03	0.66	77%	171.72
sjeng	0.81	0.68	17%	141.49
bzip2	0.94	0.49	82%	114.48
gromacs	1.35	0.54	96%	88.76
astar	0.91	0.17	62%	36.07
hmmer	1.00	0.13	97%	34.20
h264ref	0.71	0.16	81%	28.03
namd	1.09	0.09	91%	23.25

Table 3.3: Multi-programmed workloads composition. Workloads are then replicated to have 8 programs per mix. RBH : Row-buffer hit rate.

	Memory Intensive		Memory non-intensive	
Workload	High RBH	Low RBH	High RBH	Low RBH
HIGH				
H1	lbm,milc,soplex,libq			
H2	lbm,milc,libq,leslie3d			
H3	lbm,milc,soplex,leslie3d			
H4	lbm,soplex,libq,leslie3d			
H5	milc,soplex,libq,leslie3d			
MIX				
M1	milc,libq,leslie3d	mcf		
M2	lbm,milc,libq	omnetpp		
M3	soplex,leslie3d	mcf,omnetpp		
M4	lbm,libq	mcf,omnetpp		
M5	milc,leslie3d	mcf,omnetpp		
M6	soplex,milc	mcf,omnetpp		
M7	libq,soplex	mcf,omnetpp		
LOW				
L1		mcf		
L2		mcf,omnetpp		
L3		omnetpp		
LOW_BW				
LB1	leslie3d,libq,soplex		bzip2	
LB2	milc,leslie3d	omnetpp	sphinx3	
LB3	milc,soplex		sphinx3	sjeng
LB4	lbm	mcf	bzip2	sjeng
LB5	libq	omnetpp	bzip2	sjeng

Metrics

To measure system throughput, I use Weighted Speedup (WS) [67], as defined by Equation 3.1. IPC_{alone} and IPC_{shared} are the IPC of an application when it is run alone and in a mix, respectively. The number of applications running on the system is given by N.

$$WS = \sum_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{3.1}$$

System power efficiency, expressed in terms of throughput (WS) per unit power, is also reported. System power is the combined power consumption of cores, caches, and DRAM. Since I adopt a multi-programmed simulation environment,I report power efficiency, rather than energy efficiency. Each application starts executing at its predetermined execution point. I simulate the mix until the slowest application in the mix executes the desired number of instructions. Statistics per application are gathered only until each core (application) reaches the fixed number of instructions. However, I keep executing the faster applications to correctly simulate the contention for shared resources. When IPCs are compared, the IPCs for the same number of instructions across different configurations are used. Consequently, the same application in a different configuration can complete a different amount of work. For this reason, it is difficult to make fair energy comparisons across configurations; I compare power efficiency instead.

I also use minimum speedup (i.e. maximum slowdown) to determine the fairness of the system [37]. The minimum speedup of a workload is defined by Equation 3.2

$$MinimumSpeedup = \min_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{3.2}$$

The minimum speedup of a workload helps to identify whether a technique improves overall system throughput by improving the performance of some applications significantly while degrading the rest. The harmonic mean of speedups is also widely used as a system fairness metric; I favor the minimum speedup, however, to highlight the disadvantage that a program can suffer due to each approach.

3.4.2 Row-buffer Hit Rate

I first present the effect of bank partitioning on row-buffer hit rate. Figure 3.8 shows the row-buffer hit rate of each benchmark with varying configurations. The per-core (and thus per-benchmark) row-buffer hit rate are gathered and averaged to the row-buffer hit rate for the same benchmark across different workloads. For most benchmarks, the row-buffer hit rate improves with bank partitioning. The most improved benchmark is `libquantum`, which recovers half of the locality lost due to inter-thread interference. On the other hand, the row-buffer locality of `lbm` is degraded. This may be surprising considering that `lbm` has a very high row-buffer hit rate (98%) when run alone. The spatial locality of `lbm` is sensitive to the number of banks. With

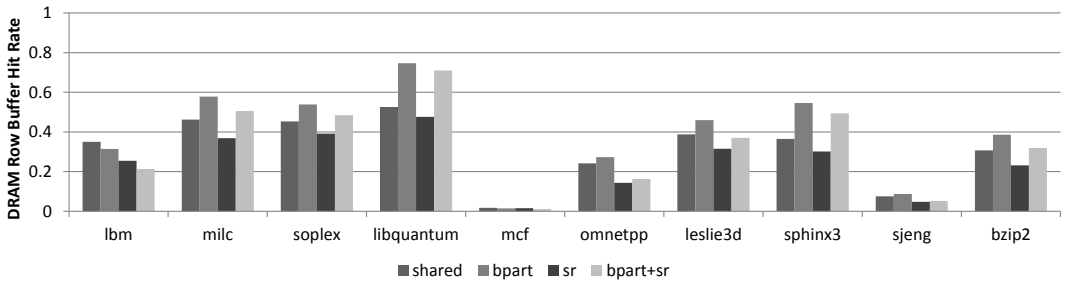


Figure 3.8: Average DRAM row-buffer hit rate of each benchmark.

a reduced number of banks due to bank partitioning, `lbm` cannot keep enough concurrent access streams open and loses row-buffer locality.

With sub-ranking, the row-buffer hit rate drops since the granularity of sub-rank interleaving is 64B and the number of independent, half-sized banks that a program accesses increases. For example, two consecutive 64B accesses would normally result in 50% hit rate, but result in a 0% hit rate once sub-ranking is introduced. Note that this is a start-up cost and is amortized over the subsequent accesses if the application has sufficient spatial locality. However, when sub-ranking is applied on top of bank partitioning, the row-buffer hit rate drops slightly due to this effect.

3.4.3 System Throughput and Fairness

Figure 3.9 shows the system throughput for each memory configuration and workload. As expected, the workload group with many high spatial locality benchmarks (`HIGH`) benefits most from bank partitioning. These benchmarks recover the locality lost due to interference and their increased row-buffer hit rate results in improved throughput. Since many of the bench-

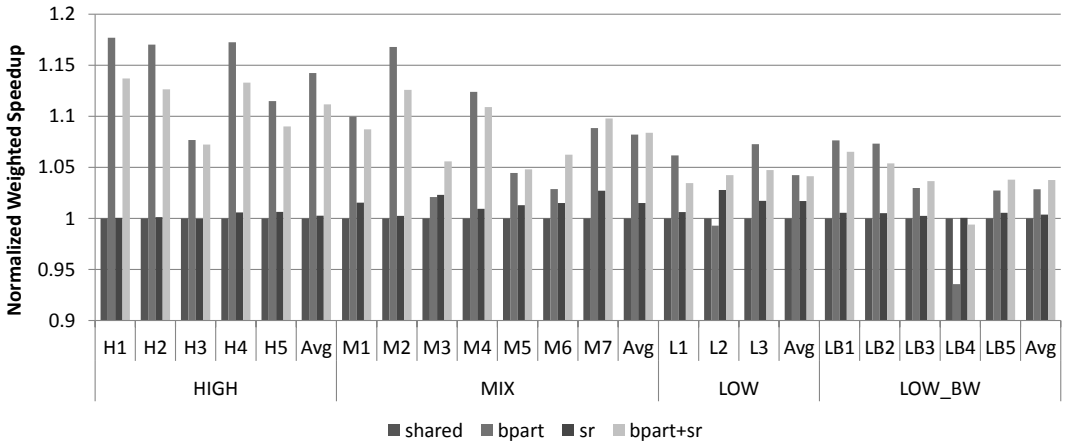


Figure 3.9: Normalized throughput of workloads. Y-axis starts at 0.9 to better visualize the differences.

marks in group `HIGH` have high spatial locality, they do not gain much from the additional banks that sub-ranking provides. Sub-ranking alone does not affect (or slightly improves) performance for the `HIGH` benchmarks, but it degrades the improvements of bank partitioning when both are applied. With shared banks, inter-thread interference results in many long-latency row-buffer misses. Additional latencies due to sub-ranking are insignificant compared to row-buffer miss latencies (4 cycles vs. 37 cycles). With bank partitioning, however, many requests are now row-buffer hits and take little time (11 cycles). Therefore, the extra latency from sub-ranking becomes relatively more costly and affects performance.

Workload group `MIX` enjoys less benefit from bank partitioning since it includes benchmarks with low spatial locality. `MIX` is also moderately improved by sub-ranking. When both techniques are applied together, however, most `MIX` workloads show synergistic improvement. Exceptions to this observation

include workloads M1, which has only one application with low spatial locality to benefit from sub-ranking, and M4, whose performance with sub-ranking is slightly degraded due to the presence of `lbm` (for reasons described above). M2 suffers from both afflictions, and its performance relative to bank partitioning degrades the most from the addition of sub-ranking, though it is still within 3% of the maximum throughput.

Workload group `LOW` shows interesting results. Intuitively, workloads composed of only low spatial locality benchmarks would suffer performance degradation with bank partitioning due to reduced bank-level parallelism. However, benchmarks L1 and L3 show a healthy 5% throughput improvement from bank partitioning. This is because bank partitioning load-balances among banks. As most of the requests are row-buffer misses occupying a bank for a long time waiting for additional precharge and activate commands, the banks are highly utilized. Therefore, all the bank-level parallelism the DRAM system can offer is already in use and bank load-balancing becomes critical for the DRAM system throughput. Figure 3.10 shows the average queueing delay of requests at the memory controller scheduling buffer. When banks are partitioned among low spatial locality threads, the requests are evenly distributed among banks, lowering the average queueing delay.

Workloads in group `LOW_BW` contain non-memory-intensive benchmarks, thus the collective impact on throughput is smaller but has a similar trend. Workload LB4 is an outlier that shows a 7.6% throughput drop from bank partitioning. It consists of `lbm`, `mcf`, `bzip2`, and `sjeng`, where `lbm` and `mcf`

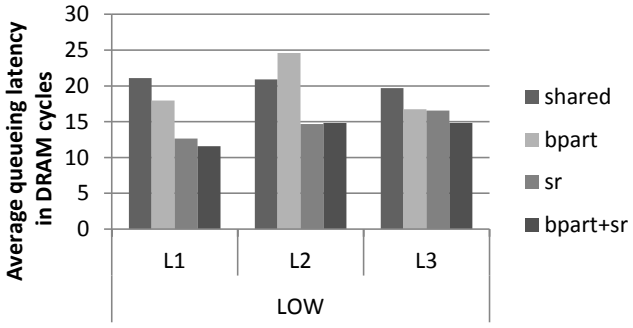


Figure 3.10: Average queueing delay of requests for workload group `LOW`

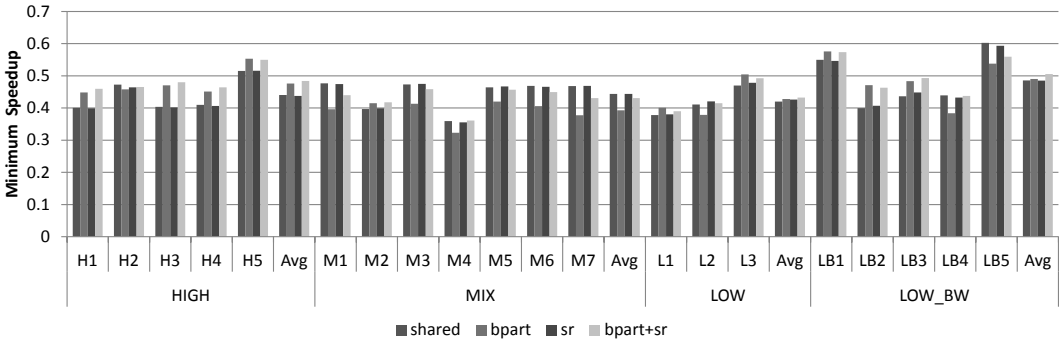


Figure 3.11: Minimum speedup of workloads.

are ill-affected by bank partitioning as shown in Figure 3.8; `bzip2` and `sjeng` are non-memory-intensive and, thus, remain unaffected. Altogether, no improvement results from bank partitioning for this workload. However, memory sub-ranking recovers some of the lost bank-level parallelism and performance. Bank partitioning combined with sub-ranking gives average throughput improvements of 9.8%, 7.4%, 4.2%, and 2.4% for `HIGH`, `MIX`, `LOW`, and `LOW_BW`, respectively. The maximum throughput improvement for each group is 12.2%, 11.4%, 4.5%, and 5%.

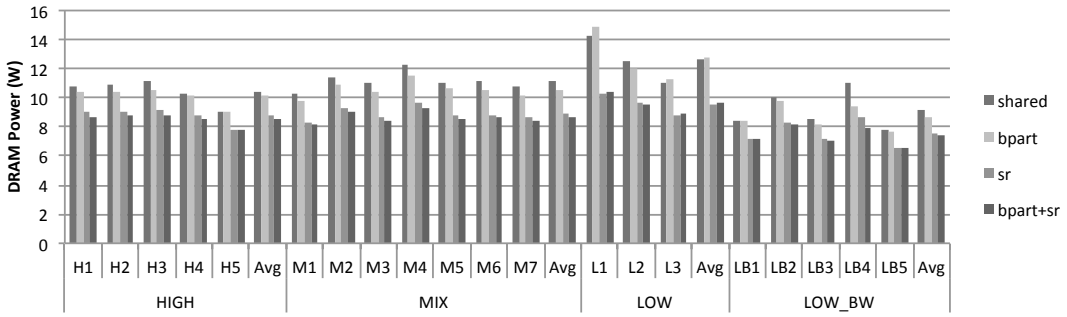


Figure 3.12: DRAM power consumption.

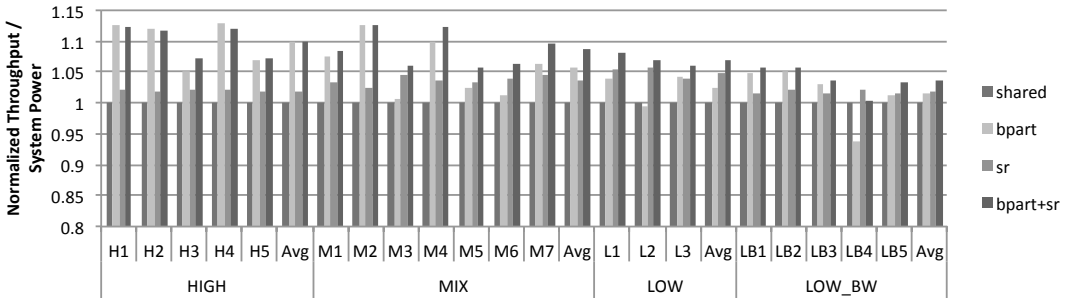
Workload throughput alone only shows half of the story. Figure 3.11 shows the minimum speedup of workloads. Although bank partitioning alone could provide almost all of the system throughput improvements, it does so at the expense of benchmarks with low spatial locality. Workloads in group `MIX` and LB4 and LB5 in group `LOW_BW` show a significant reduction in minimum speedup when bank partitioning is employed. In all such cases, the benchmark which suffers the lowest speedup is either `mcf` or `omnetpp`; both benchmarks have low spatial locality. By partitioning banks, these programs suffer from reduced bank-level parallelism. When sub-ranking is employed along with bank partitioning, the minimum speedup recovers from this drop in addition to providing system throughput improvements. Bank partitioning and memory sub-ranking optimize conflicting requirements, locality and parallelism, and employ them together results in a more robust and fairer solution.

3.4.4 Power and System Efficiency

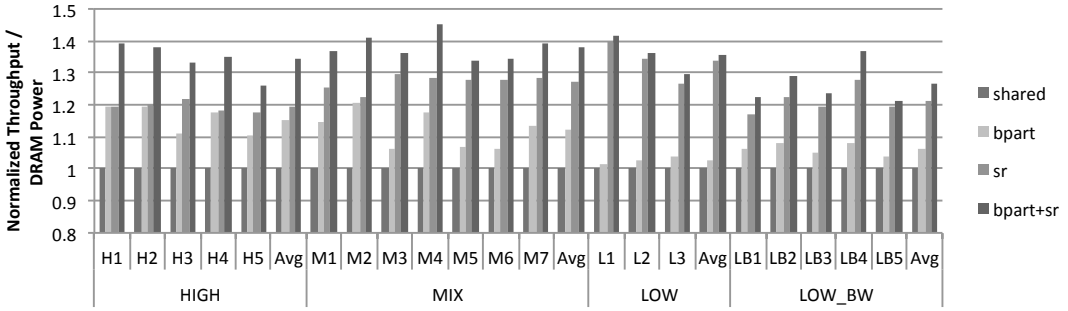
As described in Section 3.4.1, I estimate the system power by modeling the DRAM power and the power consumed by the processor. Figure 3.12 shows the DRAM power component. Although the DRAM power reduction due to bank partitioning only appears to be modest, the actual energy reduction is significantly greater since bank partitioning greatly reduces activations by improving the row-buffer hit rates, and thus performance, as shown in Figures 3.8 and 3.9. Sub-ranking reduces DRAM power consumption by only activating a fraction of the rows. Naturally, bank partitioning and sub-ranking, combined, brings additive benefits. With both schemes together, the DRAM power consumption is reduced dramatically, averaging at a 21.4% reduction across all benchmark mixes.

System efficiency is measured by throughput (WS) per system power (Watts), where system power includes DRAM and CPU power. Since the dynamic power consumption of the CPU is modeled as a function of IPC (Section 4.3), the CPU power slightly increases with the improved IPC. However, the combined system power (DRAM+CPU) remains fairly constant across benchmark mixes.

Figure 3.13(a) shows the normalized throughput per system power. In group `HIGH`, most of the benefit comes from bank partitioning alone. As mentioned earlier, this group has mixes of benchmarks that all have high row-buffer hit rates and does not benefit much by sub-ranking. On average, the improvement in system efficiency is 10%. In group `MIX`, the overall system efficiency is



(a) Normalized throughput per system power.



(b) Normalized throughput per DRAM power.

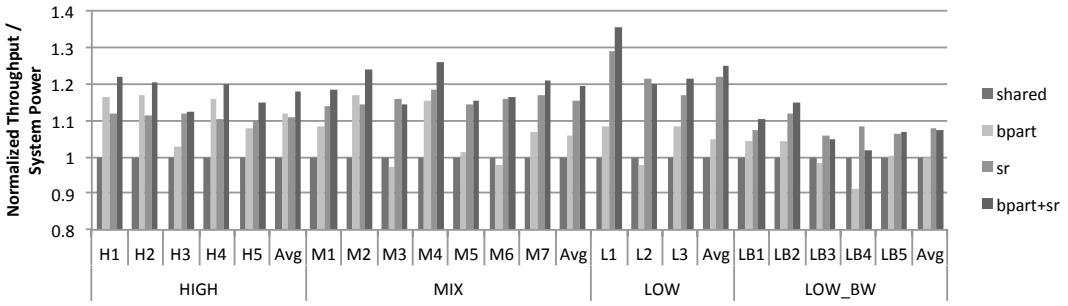
Figure 3.13: System efficiency of two-rank systems.

best enhanced by the synergy of bank partitioning and sub-ranking. This is the result of the throughput improvement from bank partitioning combined with the dramatic power savings from sub-ranking. Efficiency improves by 9% on average, whereas bank partitioning and sub-ranking alone improve efficiency only 6% and 4%, respectively. In group `LOW_BW`, although the improvement is relatively lower, using bank partitioning and sub-ranking together still brings the best benefit. Workload LB4 again shows degraded power efficiency with bank partitioning because of its impact on throughput for this workload.

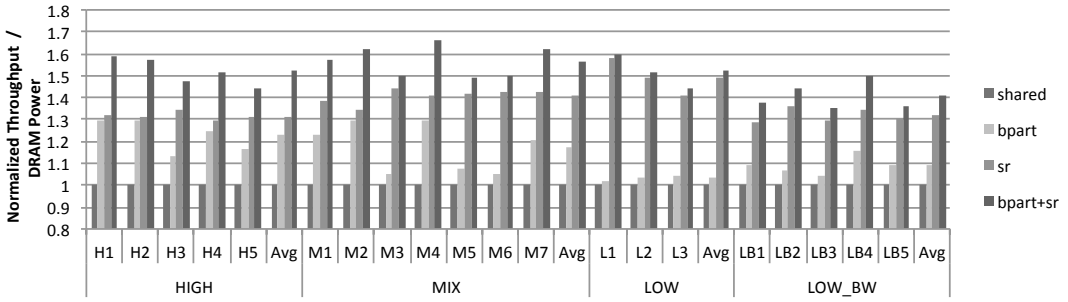
Current processor and memory scaling trends indicate that DRAM power is a significant factor in system power, especially in servers with huge DRAM capacities [23]. Such systems will see greater improvement in system efficiency with combined bank partitioning and sub-ranking. Figure 3.13(b) verifies this trend with the normalized throughput per DRAM power. Only taking DRAM power into account, bank partitioning with sub-ranking shows improvements of up to 45% over the baseline.

3.4.5 Bank-limited Systems

Processor and memory scaling trends also indicate that CMP core count increases faster than available memory banks in the system, especially for embedded or many-core systems. To approximate a system with limited memory banks per core, I conduct experiments on a system with a single rank per channel instead of the two ranks previously shown. Figure 3.14(a) illustrates the system efficiency for this bank-limited machine. The results show that the



(a) Normalized throughput per system power



(b) Normalized throughput per DRAM Power

Figure 3.14: System efficiency of one-rank systems.

bank-limited system enjoys greater benefit from bank partitioning with sub-ranking. The average improvements in system efficiency on the bank-limited system are 18%, 19%, 25%, and 7% for groups `HIGH`, `MIX`, `LOW` and `LOW_BW`, respectively. Figure 3.14(b) shows the system efficiency per DRAM power for a single rank system; combining bank partitioning and sub-ranking again provides significant improvements.

3.5 Spatial Locality in Future Systems

The number of cores in a processor is expected to keep growing in the future. To that end, some early research have investigated the feasibility of

a processor with thousands of cores [36]. On the other hand, the number of banks will grow slowly as we discussed in 3.2.2. We are already seeing many-core systems that have limited banks. Figure 3.15 shows the number of banks per hardware thread for some commercial systems availabie in 2011. Many systems with moderate memory capacity don't have more than 8 banks per thread, and some are approaching to 1 bank per thread ratio. With the scaling gap between the number of memory banks and transistors, future many-core systems will have even lower memory bank/core ratio, possibly less than one per core. The proposed spatial locality preservation scheme, realized through partitioning of banks among cores, would seem to break down in such systems.

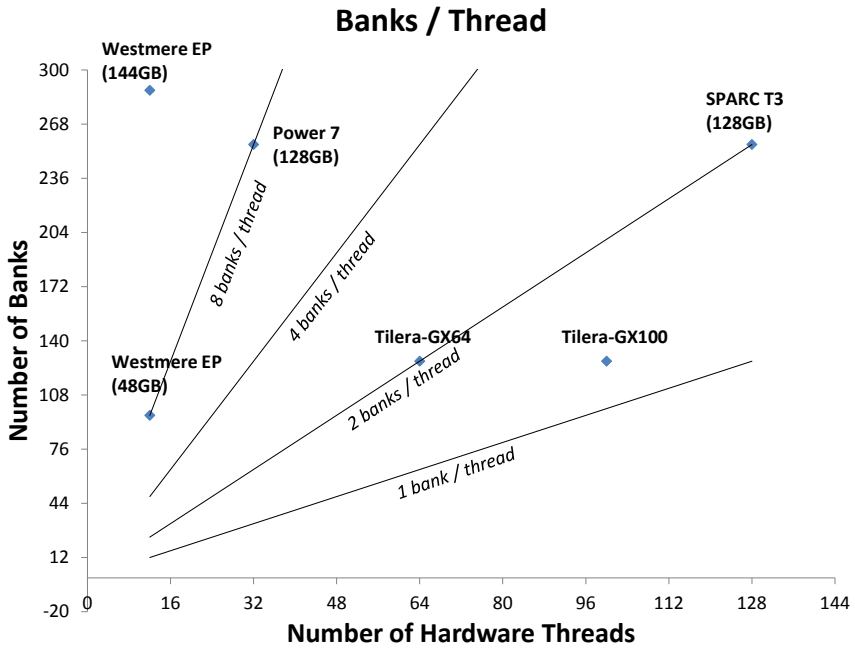


Figure 3.15: The number of banks per hardware thread in commercial systems. Memory was specced to the maximum capacity, otherwise noted.

Udipi et al. shares this view and proposes to forgo spatial locality in main memory altogether [72].

I believe such simplistic extrapolation fails to consider the differences between the cores in current multi-core processors and the future many-core processors, as well as the type of workloads running on them. The extra cores will be used to run many parallel threads from multi-threaded applications, not to consolidate thousands of independent processes to a single chip. As I will argue below, with the multi-threaded applications, the increased core count does not proportionally grow the number of competing memory access streams.

Cores in a future many-core processors will be similar to today's Niagara [57], Tilera [16], Larabee [64], and GPGPUs. Each core is less powerful than one in multi-core processors like Sandy Bridge, but is area and energy efficient so that many of them can turn parallelism in applications into performance. Threads in these multi-threaded applications share data and synchronize with each other, unlike independent, free-flowing processes in multi-programmed workloads. Memory accesses from cooperating threads are often coordinated as a performance optimization to better utilize the expensive off-chip memory bandwidth and maximize on-chip data sharing [19, 59]. In GPGPU programs, for example, memory accesses to bring in data for multiple threads are coordinated together to turn them into one large coalesced access [9]. In ray tracing, coherent rays that traverses the nearby space are scheduled together to maximize data reuse [56]. As a result, memory ac-

cesses from multiple cores traversing coherent rays show locality. As future many-cores will have higher compute/bandwidth resource ratio than today, orchestrating memory accesses will become even more important to fully utilize on-chip compute resources. Therefore, if the future system is to be used effectively, the number of independent, interfering memory access streams should not scale with the number of cores. Nonetheless, coordinating all accesses for thousands of threads will be challenging and costly in many applications. The interferences from multiple memory access streams will grow in check, from which spatial locality can be preserved and exploited in the main memory through careful isolation as proposed.

There is very little merit in using the future many-core to consolidate multi-programmed workloads to a single chip. Provisioning the memory capacity and bandwidth to support thousands of independent data set for *a single chip* is expensive both in cost and performance. Multi-processor systems can have higher aggregate memory capacity and bandwidth cost efficiently, and therefore will serve such workloads better.

3.6 Comparison of Alternative Address Mappings

Kaseridis et al. [35] proposed the Minimalist open-page policy, which advocates an address mapping scheme that trades off reduced spatial locality in row-buffers for enhanced bank-level parallelism. This mapping intentionally limits the row-buffer locality of a single thread by interleaving banks every four 64B cache lines. With such a mapping, sequential accesses switch banks

every four accesses. The motivation is to prevent a thread with high spatial locality from occupying a bank for a long period of time, delaying other threads. Figure 3.16 shows the Minimalist address mapping along with the baseline mapping I used. Minimalist maps four consecutive 64B cache lines to the same row of the same bank, and interleave across the channels, banks, and ranks every four cache lines. The interleaving order: channels, banks, and ranks, is identical to the baseline and in accordance to the order discussed in Section 2.2.2. Under this minimalist mapping, a 4KB page is now spread over 16 banks across two channels. Unless a thread presents an access pattern with a stride of 16KB in the physical address space, it would not send more than four consecutive accesses to the same row. In comparison, the baseline mapping maps a 4KB page over only 2 banks across two channels. With a sequential access pattern, a thread can occupy a bank for up to 32 accesses for a 4KB page (4KB / 2 banks / 64B accesses).

Finer interleaving granularity of the Minimalist mapping also affects its row-buffer management policy. Since only four consecutive cache lines are mapped to the same row, rows left open are unlikely to get any future row-buffer hits. Therefore, a close-page policy should be used to save precharge delay for future accesses to different rows.

Minimalist mapping essentially takes an opposite approach to bank-partitioning which tries to preserve the row-buffer locality of a thread as much as possible. Interestingly, bank-partitioning also addresses the very problem the Minimalist mapping tries to solve, i.e., a thread with high spatial locality

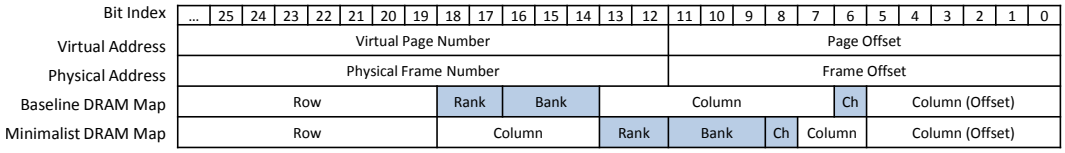


Figure 3.16: Comparisons of baseline and Minimalist address mapping.

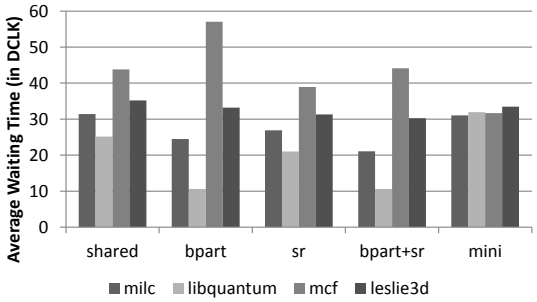


Figure 3.17: Average time a request waits in the scheduling queue until the target row is activated and ready to accept column accesses. If the row was already open when a request arrived, its waiting time is 0.

delaying other threads by occupying a bank for a long period of time. Because banks are exclusive between threads, a thread never have to wait for other threads for a bank. As both Minimalist and bank-partitioning solve the long-occupancy problem through address mapping scheme, the scheduler can arbitrate between threads simply based on its optimization goal whether it is system throughput or fairness [35, 37, 38, 54, 58]. Bank-partitioning achieves this isolation at the cost of reduced bank-level parallelism per thread, whereas the Minimalist mapping sacrifices row-buffer locality. As a result, bank-partitioning benefits high locality benchmarks and Minimalist benefits low locality benchmarks more, respectively. I will show how these mechanisms impact memory access characteristics of applications differently.

Figure 3.17 shows queueing delays of memory accesses of the workload M1 from group `MIX` for different configurations, including the minimalist mapping. Cores {0,4}, {1,5}, {2,6}, and {3,7} run `milc`, `libquantum`, `mcf`, and `leslie3d`, respectively. Benchmark `libquantum` has the highest spatial locality and therefore the shortest queueing delay because requests do not have to wait for the row to be activated. Benchmark `mcf` has the lowest spatial locality and thus longest queueing delay in this workload. When a bank partitioning is used in (`bpart`), the queueing delay of `libquantum` reduces from 24.5 to 10.6 dclk. As previously open rows can remain in the row buffer, due to the interference isolation, the subsequent requests do not have to wait for the row to be activated. Benchmark `mcf`, however, has to wait longer (57.0 dclk) because of reduced bank-level parallelism. When a sub-ranking is added (`bpart+sr`), the average wait time of `mcf` cores decreases to 44 dclk. The minimalist mapping (`minimalist hash`) makes the queueing delay insensitive to application spatial locality. All four benchmarks wait roughly the same time in the queue, 30.4 dclk on average. Compared to other configurations, `milc` and `libquantum` wait longer for the row to be activated, but `mcf` wait less because the row-miss requests wait less for precharges due to the close-page policy.

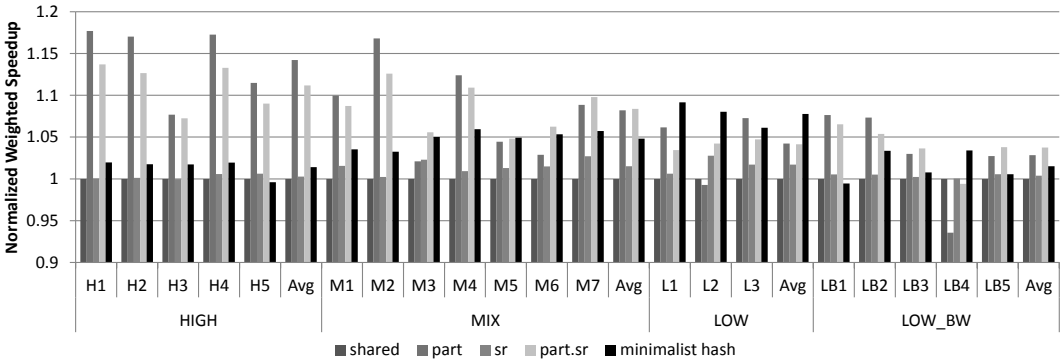
3.6.1 System Throughput and Fairness

Now we compare the impact of different address mappings in system throughput and fairness over a variety of workloads. In their work, Kaseridis

et al. [35] also proposed a priority scheme based on each thread's memory-level parallelism (MLP) on top of the address mapping scheme and showed a significant additional improvement in system throughput and fairness. Since our discussion is on address mappings, this priority scheme is not included in results presented here. I leave studying the interaction between address mappings with different priority schemes as future work.

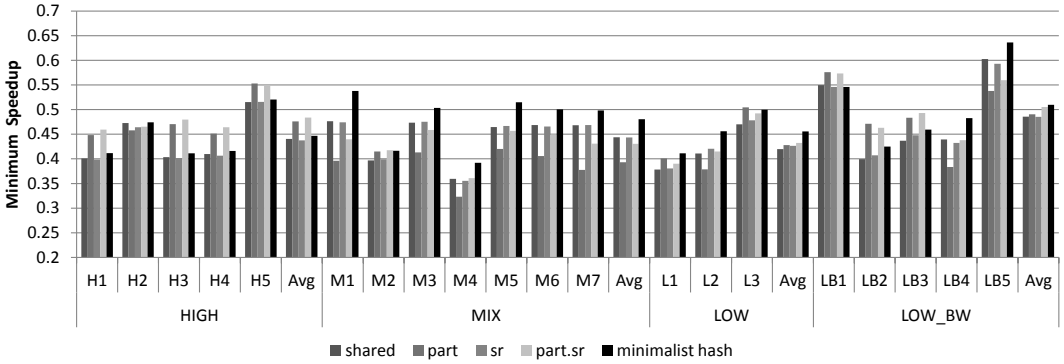
Figure 3.18 shows the system throughput of Minimalist mapping (`minimalist hash`) with `shared`, `bpart`, `sr`, and `bpart+sr`. As expected, workloads with high spatial locality (`HIGH`) benefits more by bank-partitioning (`bpart`) and workloads with low spatial locality (`LOW`) benefits more by Minimalist mapping. Eliminating the long-occupancy by the Minimalist mapping improves the throughput of `HIGH` a little (1.5%), but the additional benefit of preserving the row-buffer locality is very effective (14%). Workload group `LOW` is improved the most by Minimalist mapping (7.7%) due to higher bank-level parallelism and latency saving from close-page policy precharging the bank for future row-miss accesses. When open-page policy was used with Minimalist mapping, throughput improvement reduced to 3.5%.

Figure 3.19 shows the fairness metric of minimum speedup. For workload groups `MIX`, `LOW`, and `LOW_BW`, Minimalist mapping shows the largest fairness improvement among the configurations. Minimum speedup of those workload groups is determined by the performance of low spatial locality benchmarks, because they slow down more under bandwidth contention than high spatial locality benchmarks, whose accesses are favored by scheduler for band-



(a) DDR3-1600 with 64B channel interleaving

Figure 3.18: Normalized throughput of workloads. Y-axis starts at 0.9 to better visualize the differences.



(a) DDR3-1600 with 64B channel interleaving

Figure 3.19: Minimum speedup of workloads

width efficiency. With Minimalist mapping benefits low locality benchmarks by improving bank-level parallelism and employing close-page policy, fairness of workloads improve as a result. The fairness can be further improved when the MLP-based priority scheme is used at the same time, which was reported as 7.5% on average and 15% maximum [35]. For workload group `HIGH` which consists of high locality benchmarks only, improvement of Minimalist mapping is

limited as we have seen in the throughput results. Bank-partitioning (`bpart` and `bpart+sr`) improves the high spatial locality benchmarks and result in higher minimum speedups for workload in `HIGH`.

3.6.2 Address Mapping and Memory Bandwidth

As shown in previous section, interleaving granularities across DRAM structures have significant impact on memory system throughput and fairness. Small granularity increases parallelism, e.g. cache-line interleaving across channels or four cache-line interleaving across bank by Minimalist, and large granularity prioritizes locality, e.g. row-interleaving across banks by baseline. The optimal balance point between the parallelism and locality changes depending on the core performance and memory system bandwidth. In this section, I evaluate different interleaving granularities for channels and banks for memory systems with different bandwidth [3]. The results show that the best configuration changes with the memory bandwidth available for *each* core or the relative core performance and available bandwidth.

Cache-line channel interleaving, as in our baseline, is widely used in modern memory systems [1, 6]. Since low-order address bits are likely to change more frequently than high-order bits, using fine-grained interleaving help balancing loads on channels. Such mapping also enables a single thread to use the bandwidth of all channels simultaneously. Figure 3.20 shows the per-

[3]Ranks can simply be considered as banks that are more expensive to switch, and rows must have the coarsest interleaving granularity as it does not offer any parallelism.

formance of single-thread benchmarks for a range of channel interleaving granularity (64B to 8KiB) and memory system bandwidth (DDR2-533, DDR3-1066 and 1600). The rest of the system parameters, including the performance of a

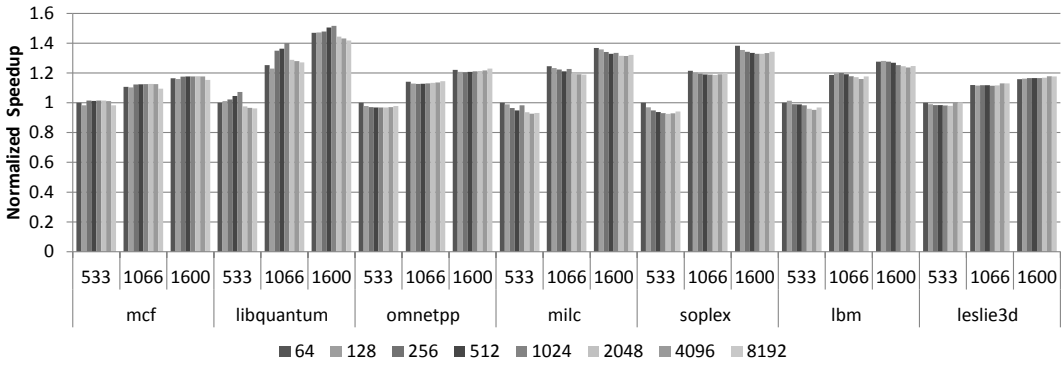


Figure 3.20: Speedup of benchmarks with varying interleaving granularity (64B to 8KiB) and memory system bandwidth (DDR2-533, DDR3-1066 and 1600) over 64B/DDR2-533 configuration. Benchmarks insensitive to the varied parameters are not shown.

core, are kept identical. Interestingly, single-thread performance did indeed not improve much over the memory system generations we are considering. The DDR2-533 parts were used with Intel Prescott unicore processor which was clocked up to 3.8GHz in 2004 [28], and the current DDR3-1600 parts are used with Intel Ivy Bridge processor whose maximum frequency is still 3.9GHz [6]. Figure 3.20 shows two findings. First, as expected, the finest interleaving granularity (64B) shows the best performance for most benchmarks, up to 8% for `soplex` in DDR2-533 over 2KiB granularity. [4] Second, the performance advan-

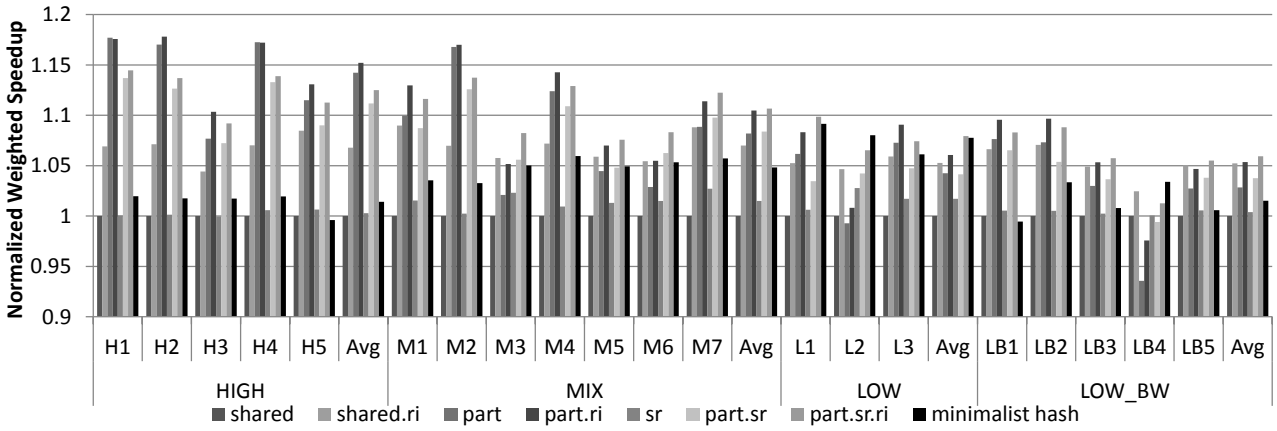
[4] `libquantum` performs the best at 1KiB interleaving granularity. It has phases in which arrays are read and written serially. When fine grained interleaving is used, both channels often go into the write mode in which reads are blocked. When coarser grained interleaving

tage of fine-grain interleaving is more prominent in lower-bandwidth memory systems. As memory bandwidth grows while the performance of a core remains the same, a single channel bandwidth has become sufficient for a single thread and channel interleaving adds little. Still, when only a single thread performance is considered, fine-grain channel interleaving works the best. However, when multiple cores share the memory system, additional considerations such as efficiency and fairness are introduced.

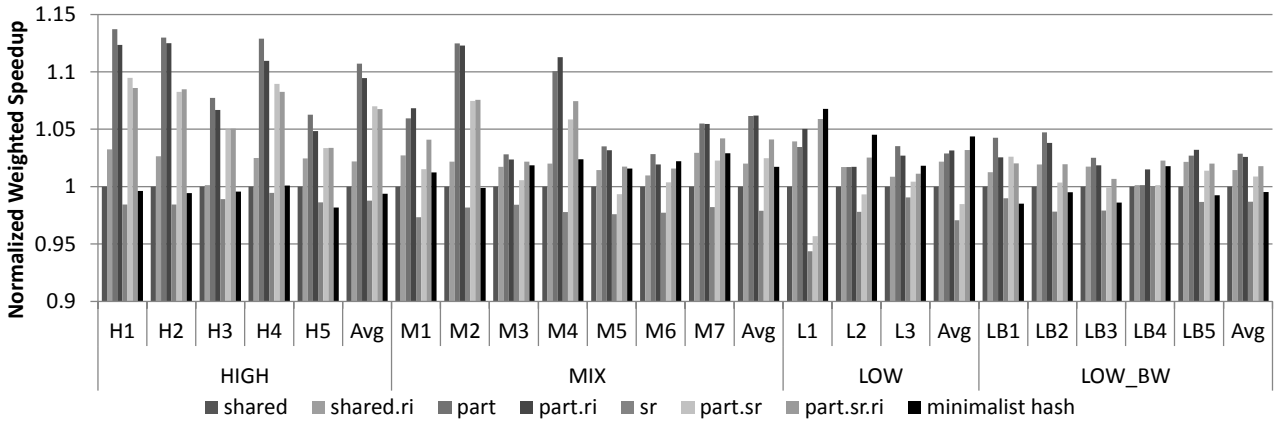
Figure 3.21 shows the system throughput results with additional configurations which interleave addresses across channels at row granularity (`shared.ri`, `bpart.ri`, and `bpart.sr.ri`). For a DDR3-1600 system, row-interleaving (`shared.ri`) clearly performs better than cache-line interleaving (`shared`) across all workload group. As we have seen in the single-core results, parallelism over channels does not bring additional benefit to this core and memory combination. Instead, channel-interleaving costs locality of streams since accesses are spread over channels. Two corresponding banks in different channels effectively form one larger bank. The number of independent banks that can hold active access streams is halved. This limits the bank-level parallelism necessary for multi-core systems. When an entire row is interleaved across channel, accesses from one stream are localized to one bank on a channel at a time, and the bank on the other channel can hold a different row for another access stream. In multi-core systems with single-channel bandwidth high enough,

is used, one channel handles streaming writes and the other channel can simultaneously serve reads.

the benefit of this inter-thread parallelism and intra-thread locality outweighs that of intra-thread parallelism. Row interleaving granularity improves the performance in combination with the bank-partitioning (`bpart.ri`) and sub-ranking (`bpart.sr.ri`). Since the sub-ranking was introduced to compensate for the bank-level parallelism, the addition by row-interleaving also helps.

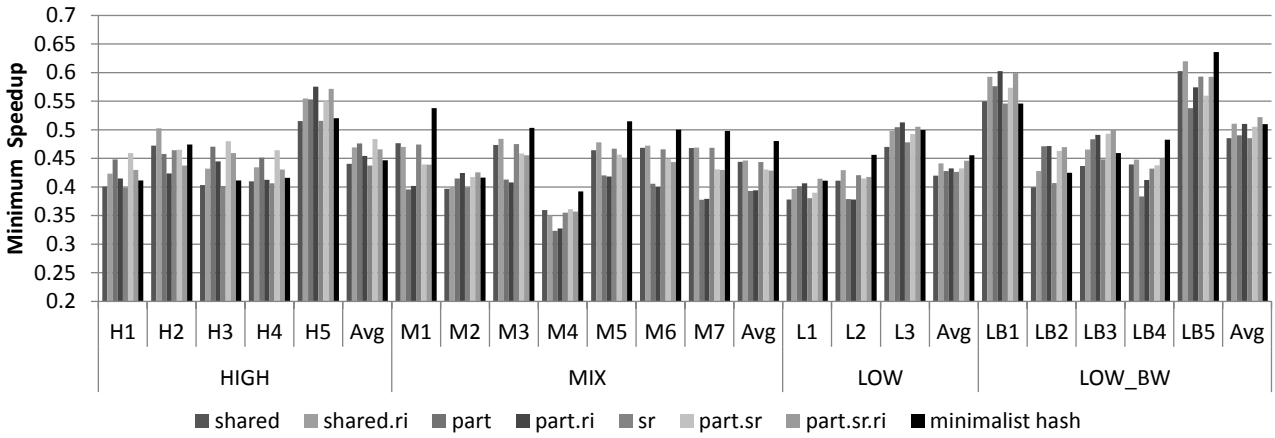


(a) DDR3-1600

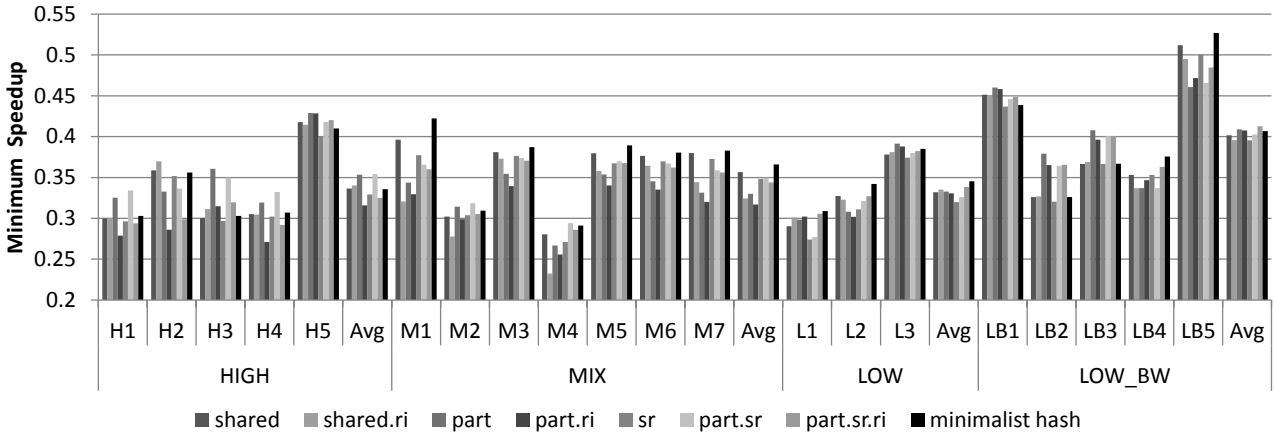


(b) DDR3-1066

Figure 3.21: Normalized throughput of workloads. Y-axis starts at 0.9 to better visualize the differences.



(a) DDR3-1600

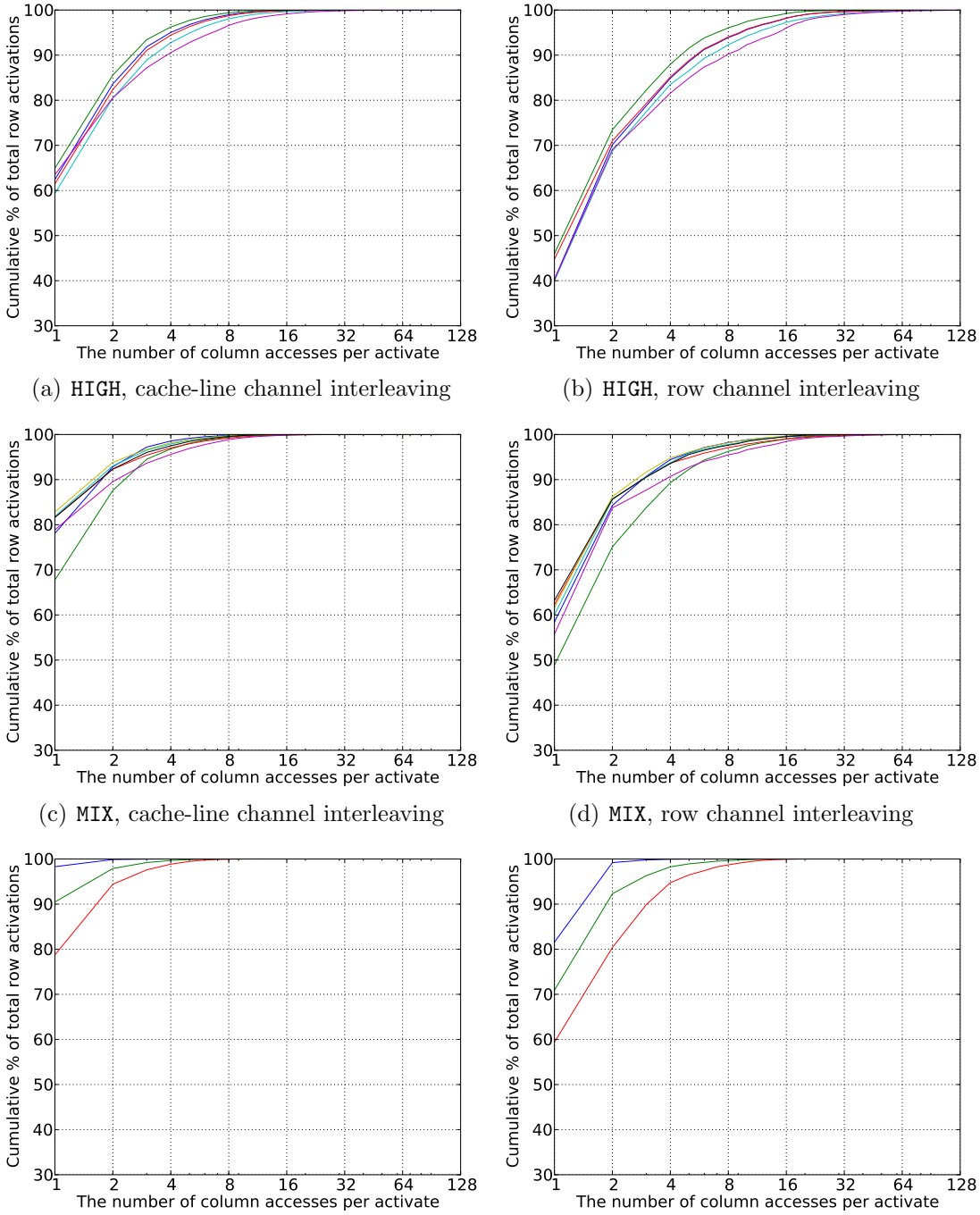


(b) DDR3-1066

Figure 3.22: Minimum speedup of workloads

The relative benefit of locality and parallelism change when lower bandwidth memory system is used, and row channel interleaving becomes less effective. Throughput increase over cache-line interleaving is reduced to 2%, compared to 6% for high-bandwidth system. As the single-channel bandwidth is reduced, the benefit of additional bandwidth from multiple channels achieved by channel interleaving is larger as we have seen in Figure 3.20. More importantly, Figure 3.22 (b) shows that fairness degrades in many workloads as opposed to improving as in a higher bandwidth results (Figure 3.22 (a)), especially for those in workload group `MIX` with maximum 14% drop for M1. Because each request takes longer, a thread occupies a bank for a longer period of time and increases queueing delay of other threads. When combined with row-interleaving, which localizes the sequential column accesses to a single bank, the duration of long burst of accesses from a thread increases significantly.

Figure 3.23 shows this increase in bank occupancy in more detail with cumulative distributions of column accesses (reads and writes) per activation: the charts on the left column (a), (c), (e) are from cache-line interleaving across channels, and the ones on the right column (b), (d), (f) are from row interleaving across channels. Both configurations were baseline shared-banks (`shared`). For a cache-line channel interleaving, only 6%, 3%, and 1% of activations serve more than four column accesses before a row is closed, for workload groups `HIGH`, `MIX`, and `LOW`, respectively. Due to the interference from other threads, most of the rows are closed after a few accesses. This is a different



(a) `HIGH`, cache-line channel interleaving

(b) `HIGH`, row channel interleaving

(c) `MIX`, cache-line channel interleaving

(d) `MIX`, row channel interleaving

(e) `LOW`, cache-line channel interleaving

(f) `LOW`, row channel interleaving

Figure 3.23: Distribution of column accesses per activate. Each line represent a workload. See Table 3.3 for the workload composition.

representation of the same data that motivated the bank-partitioning (in Figure 3.1), showing the reduction of row-buffer hit rates when the benchmark `lbm` run with other applications. On the other hand, 15% of activations serve more than four column accesses for workload group `HIGH` when row channel interleaving is used. Coarser channel interleaving localizes the accesses to a row, and the lower bandwidth per channel leads to each request taking longer. The net effect is a smaller gap between adjacent requests, and hence, less opportunity for bank-conflicting requests to precharge the row. As benchmarks with high spatial locality have such long burst of column accesses, low locality benchmarks have to wait long and fairness degrades.

To summarize, the relative core performance and channel bandwidth determines the optimal channel interleaving granularity. With stagnating single-thread performance and increasing single-channel bandwidth, cache-line channel interleaving does not improve single-thread performance. With increasing number of cores on a processor, on the ohter hand, the benefits from row channel interleaving, higher efficency and additional bank-level parallelism, seem to outweigh. I showed mainstream workstation systems with DDR3 memory, but similar analysis will be necessary when any of these parameters change. If the processor in question integrates more powerful cores or different type of cores such as GPUs, or more aggressive prefetcher is used, memory-level parallelism from a single memory access stream can significantly increases. The column access burst behavior will be more close to the systems with lower bandwidth per core system. The accumulation rate can also change

and long occupancy can be problematic. Per-channel bandwidth can be also different, for example 3d-stacked Wide-IO DRAM where there are many channels running at much slower frequency.

3.7 Summary

In this chapter, I discussed the impact of locality and parallelism in memory systems on the performance and efficiency of multicore CMP systems. Sharing of memory bandwidth among multiple cores destroys the locality within each stream and results in a inefficient use of the limited off-chip memory bandwidth. As a solution, I presented a mechanism that can fundamentally solve the problem of inter-thread locality interference at its root. I proposed bank partitioning through OS-controlled coloring to assign a set of independent banks to each potentially conflicting thread. To compensate for the reduced bank parallelism available to each thread, I used DRAM sub-ranking to effectively increase the number of banks in the system without increasing its cost. Evaluation demonstrates that this combined bank partitioning and sub-ranking technique enables us to significantly boost performance and efficiency simultaneously, while maintaining fairness as the evaluation shows. Moreover, I show that preserving locality is even more important when considering system configurations that inherently have a small number of banks per thread. I expect future systems to have such bank-constrained configurations because of the relatively high cost of increasing memory banks compared to the cost of increasing the number of concurrent threads in the CPU.

I also discussed the trade-offs involved with various physical-to-dram address mapping schemes. Essentially, address mapping is about deciding the granularity of interleaving across various DRAM structures such as banks and channels. Fine granularity improves parallelism and fairness, while coarser granularity improves locality and efficiency. Therefore, applications with low/high spatial locality benefit from fine/coarse interleaving, respectively. The sensitivity of system performance to these parameters, however, change depending on the relative core performance and memory bandwidth. As processors integrate more cores for higher throughput rather than increasing single core performance, row-buffer locality from coarse interleaving brings more benefit than parallelism across channels from fine interleaving.

Chapter 4

Dynamic Bandwidth Management in SoCs

To both reduce cost and improve energy efficiency, an increasing number of components are being integrated onto a single chip. These *systems on a chip* (SoCs) are typically composed of multiple types of *intellectual property cores* (IP cores) with different functionality. This is done both because heterogeneity increases performance and efficiency and because IP cores decrease development time. All integrated cores share common resources, such as off-chip memory, which is often one of the most constrained resources. High end SoCs, for example, now include powerful CPU and GPU cores, and both very demanding of the memory system. Fairly allocating the scarce data between the CPU and GPU cores, which have very different requirements, is both challenging and important. The CPU is latency sensitive and cannot tolerate long memory latency without losing performance. The GPU, on the other hand, is designed to tolerate long latencies but requires consistent high bandwidth for periods of time to meet its real-time deadlines. Because the CPU is sensitive to latency, it is common practice to always prioritize requests from the CPU over those of the GPU. I show that such a static policy can degrade GPU performance and result in an unacceptably low frame rate when CPU monop-

olizes the shared memory service. Conversely, prioritizing GPU requests can significantly degrade CPU performance without GPU performance gain.

In this chapter, I propose a new mechanism to solve this challenge by dynamically adjusting the memory controller's quality-of-service (QoS) policy [30]. As is done today, CPU requests are prioritize by default and GPU requests are serviced opportunistically. When the GPU is expected to miss a deadline, however, the GPU service rate is increased by raising its priority and allowing a larger number of outstanding GPU memory requests. The key to this technique is identifying when the default CPU-priority policy should be adjusted. I proposed to utilize knowledge of the GPU architecture and monitor the progress of processing a frame against the frame deadline. The memory controller can than determine when a deadline is likely to be missed and boost the GPU service quality. Although this chapter discusses a specific case with a CPU and a GPU, but this deadline-aware strategy can be applied more broadly when deadline-driven and best-effort components share a constrained resource.

4.1 Background

This section briefly compares the memory access and execution characteristics of CPU and GPU cores, as well as the fundamental principles and design of modern memory controllers and QoS mechanisms.

4.1.1 CPU

Modern general purpose processors are designed mainly to maximize the performance of a single thread of execution. Single-thread performance is very sensitive to long-latency memory requests because instructions dependent on the long latency load cannot proceed until the load completes. Caching and out-of-order execution can mitigate the impact of long main memory latency. Main memory access latency, however, is much higher than what the out-of-order structure can tolerate and cache misses that go out to main memory inevitably stall the accessing thread [33]. Therefore, any increase in CPU memory access latency, such as delays introduced by contention from the GPU, decreases CPU performance. Another important memory access characteristics of CPU is the memory bandwidth requirement can vary significantly from one application to another. Many applications cache well and rarely access main memory. Applications that process large amounts of data, on the other hand, do access main memory frequently. Even though such applications consume significant memory bandwidth they are still fundamentally sensitive to latency. The memory controller for heterogeneous cores should be designed considering such diversity in bandwidth requirement of CPU applications.

4.1.2 GPU

In an SoC where main memory is shared between CPU cores and a mobile GPU, a frame is rendered in five steps:

1. A CPU core writes scene-description data and rendering code to shared memory
2. The GPU reads in vertices from the shared memory, transforms them, and writes them back into memory
3. The GPU reads in the transformed vertices and generates fragments (potential pixels)
4. The GPU *shades* the fragments to determine the color for each pixel and writes the pixels to the frame-buffer in main memory
5. The LCD controller reads in the frame-buffer data from main memory and displays it to the LCD

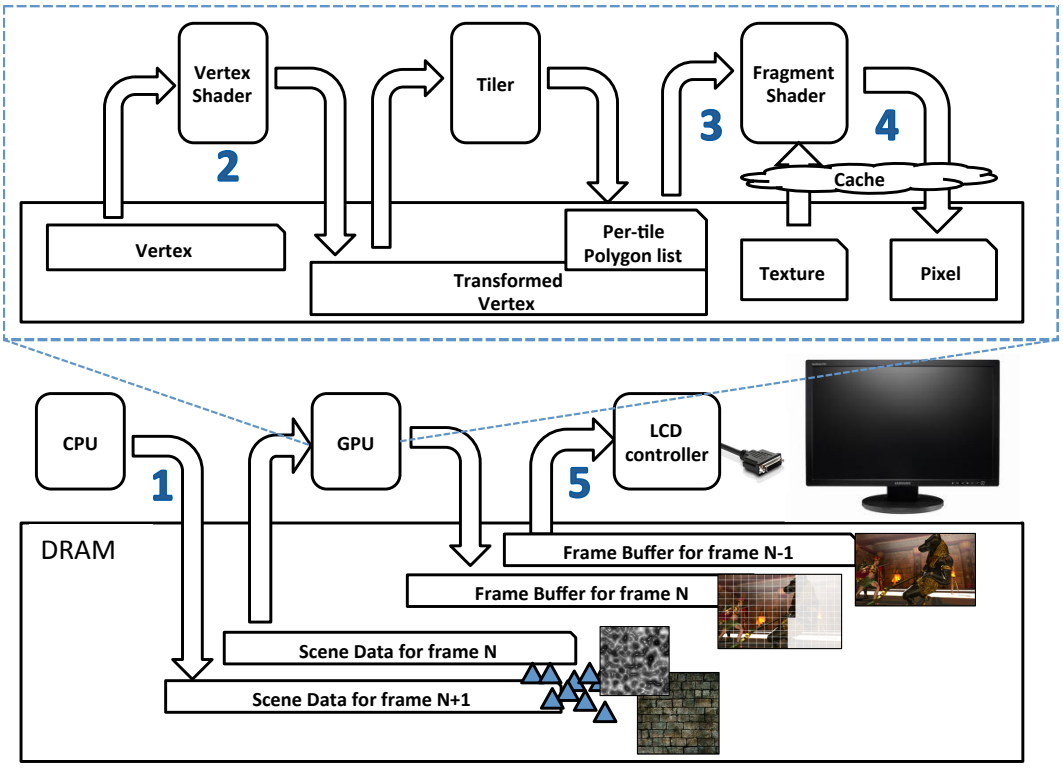


Figure 4.1: The bottom figure shows how CPU/GPU/LCD controller communicate through shared memory in a triple-buffered fashion. The upper figure shows GPU pipeline in detail.

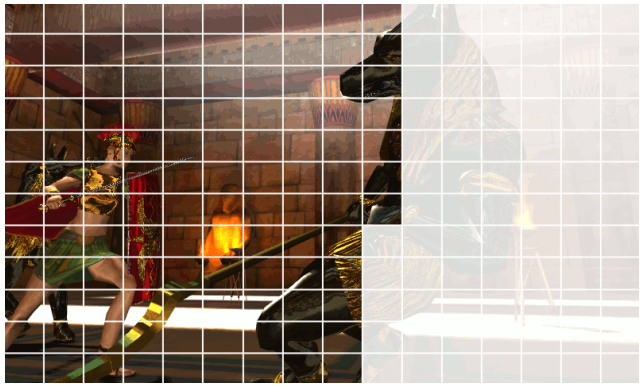
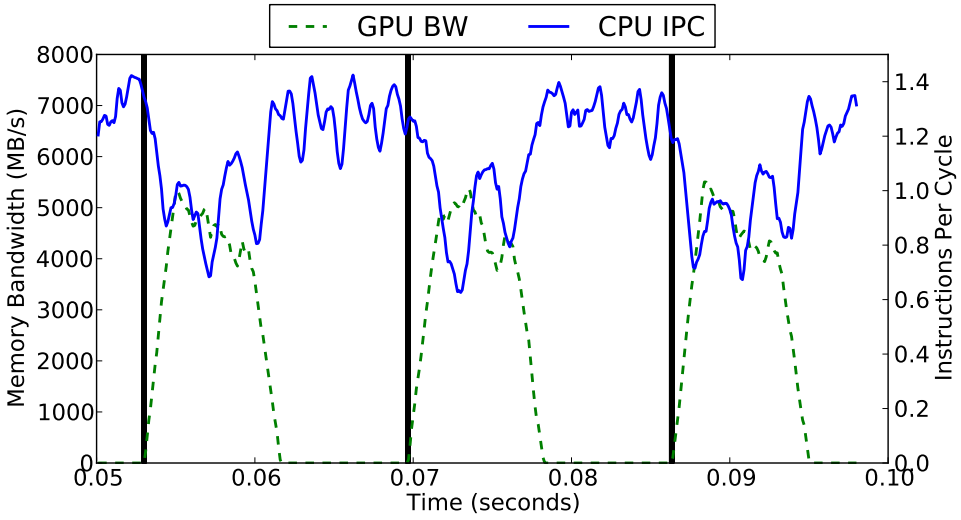


Figure 4.2: Tile based rendering in progress. Shaded tiles are to be rendered.

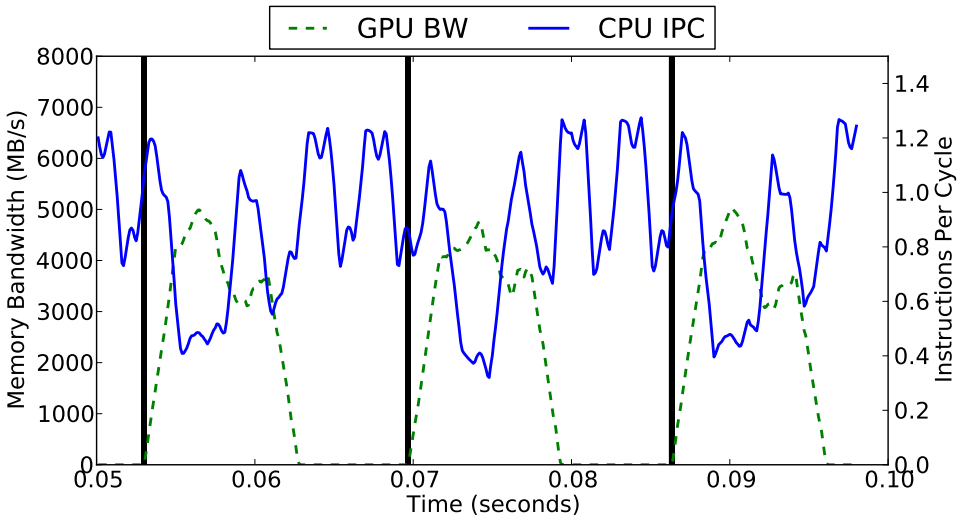
Mobile GPU cores often utilize tile based rendering to reduce off-chip memory bandwidth consumption. The screen is subdivided into many blocks/tiles, which can be processed independently of one another (Figure 4.2). As the tiles are small enough, the entire pixel data of a tile can be kept in on-chip buffers while being rendered so that repeated accesses to the same pixel do not incur off-chip memory accesses. GPUs can process all vertices and fragments within the tile and multiple tiles in parallel and can therefore tolerate very long memory latencies. They still require high bandwidth and are sensitive to disruptions in available bandwidth.

Even though only a few tile (one per core, for example) is processed at any given time, all vertices and all fragments within the tile can be processed in parallel. GPUs exploit this abundant parallelism with resultant numerous hardware threads that can be used to hide long latency memory operations. GPUs can therefore tolerate very long memory latencies, but require high bandwidth and are sensitive to disruptions in available bandwidth.

To give the illusion of motion, frames must be rendered at a certain frequency, which is typically 60 *frames per second* (FPS) or one frame every



(a) CPU cores: `mcf-art`, GPU unconstrained



(b) CPU cores: `art-art`, GPU unconstrained

Figure 4.3: GPU activity (bandwidth consumption) and CPU performance over time. Vertical lines represent frame deadlines. Experimental setup discussed in Section 4.3.

16.7ms. Humans are very sensitive to motion, and thus skipped frames (missed deadlines) are very noticeable and degrade the user experience. Scenes vary in complexity from frame to frame. Simple scenes can be processed rapidly and generate correspondingly low memory traffic. Others may take the entire time allotted or longer, resulting in skipped frames and degraded user experience. The GPU may idle between finishing a frame and starting the next frame, because frame rate is fixed and frame render time varies. Figure 4.3 shows an example of processing one frame from the Taiji GPU workload [4] with two CPU cores running together. The figure shows how the GPU only requires about half the frame time to process a scene (GPU bandwidth consumption shown with dashed lines) and consumes up to 62% of total memory bandwidth when not constrained by the memory controller as discussed below. The figure also shows how the heavy bandwidth use of the GPU hurts CPU performance (CPU's instructions per cycle (IPC) shown with solid lines) compared to when the GPU is idling. The two subfigures show different CPU workloads and the same set of GPU frames. Both `mcf` and `art` are memory-intensive applications from the MinneSPEC suite [40], with `art` requiring somewhat higher bandwidth.

4.1.3 GPU Memory Access Pattern

Graphics workloads present streaming traffic with very high spatial locality. Vertices are stored in an array and read in, transformed, and written back to another array all sequentially. Textures are accessed in a non-

sequential manner during fragment shading, but still have good spatial locality. Finally, writes to the frame buffer is also grouped with nearby pixels due to the tile-based rendering. Because of this open-page friendly access pattern, FR-FCFS scheduling can prioritizes requests from GPU over those from CPU when they compete for bandwidth. With the latency-sensitive nature of CPU execution, CPU performance suffers significantly as shown in Figure 4.3.

As discussed in Section 2.2.4, a row-hit prioritizing FR-FCFS scheduler can cause starvation of cores running low spatial locality applications. It is more problematic in heterogeneous SoCs than in general-purpose CMPs since GPU workloads can consume a lot more bandwidth for a long time with very high spatial locality. Simple starvation prevention mechanisms discussed, such as an age-based promotion, can be insufficient because constantly relying on a fall-back mechanism can cost too much CPU performance.

4.2 Quality of Service for Heterogeneous SoCs

An out-of-order memory scheduler increases overall bandwidth, but in a shared memory SoC, the priority scheme can starve some cores when other cores offer frequent requests with high spatial locality, like GPUs do. To prevent such unfairness, the memory controller must balance the accesses from different cores and provide QoS mechanisms. Because of the heavy competition in the SoC industry, very little information on how commercial SoCs manage shared memory bandwidth is publicly available.

Most previous literature on off-chip memory bandwidth QoS focuses on a different context than our multi-processor SoC (MPSoC), such as real-time systems and general purpose chip multiprocessors (CMP). This void of research has also been identified by both the real-time systems community [13] and the high-performance architecture community [15]. High-end SoCs combine both real-time and best-effort components and place very high pressure on shared memory bandwidth. Prior work on QoS or fairness for general-purpose CMPs (e.g., [35, 37, 38, 54, 55, 58, 61]) does not consider real-time constraints, thus can lead to an unacceptable rate of missed deadlines for the GPU. Work on real-time systems, on the other hand, has focused exclusively on bounding the latency of individual requests and ensuring a minimal fraction of shared throughput [12]. This approach sacrifices effective memory scheduling in favor of guaranteed deadlines, which leads to very poor utilization of available DRAM bandwidth and requires significant over-provisioning of this scarce and expensive resource.

4.2.1 Static Quality of Service

Interconnect and memory controller IPs do provide a range of QoS mechanisms, which provide control over the number of outstanding memory requests, request priority, issue rate, and target latency [7, 8]. However, literature on how to apply them properly is scarce. Recent white papers by Ashley [69] and Tune and Bruce [71] discuss general quality-of-service techniques and recommendations for heterogeneous SoCs and appear to describe

the status quo. This status quo is that two techniques are effective when combined: regulating the number of outstanding GPU requests and prioritizing CPU requests over ones from the GPU. Previous academic literature does not address this particular problem of sharing bandwidth between best-effort and real-time workloads from different cores.

Restricting the number of outstanding GPU requests reduces the GPU's ability to continuously send requests to the memory system, even though the abundant parallelism associated with graphics allows many concurrent requests. The smaller the number of outstanding GPU requests, the greater the number of memory access issue slots that are available for other cores. There are several equivalent mechanisms that can be used to constrain the number outstanding GPU requests, including separate memory controller queues for each core, which may be either physical or virtual.

Guaranteeing available request queue slots is insufficient because the memory controller may still prefer to always issue GPU requests. To avoid this situation, an age-based QoS technique can be used [54]. Recent guidelines, however, suggest that CPU requests should receive higher priority to decrease the performance lost to contention-induced high latency memory accesses [69, 71]. While the priority policy is static, the GPU may take advantage of much of the available bandwidth when CPU cores do not access main memory frequently.

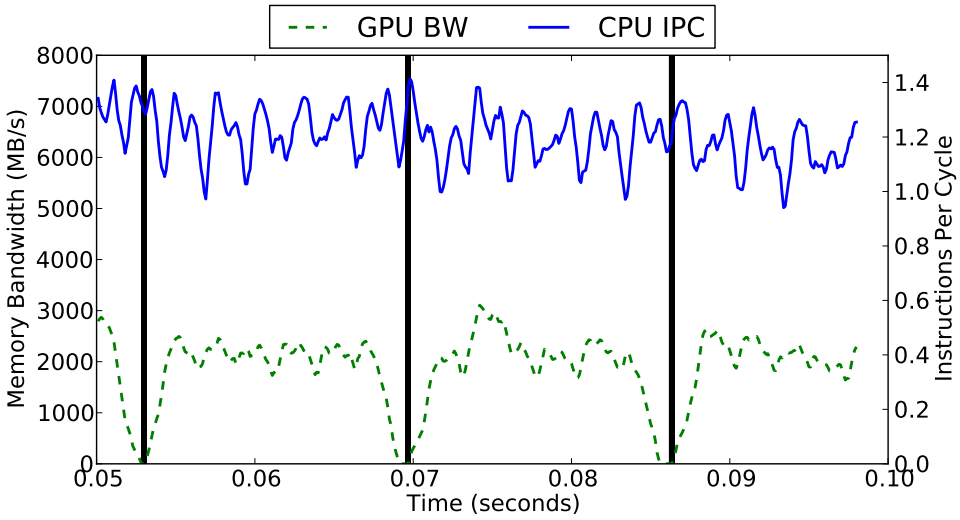
Prioritizing CPU requests indiscriminately, however, can hurt GPU performance significantly, when a CPU core continuously uses high memory

bandwidth and a GPU workload is complex enough to mandate high memory bandwidth as well. The impact of the aggressive QoS is shown in Figure 4.4. The figure shows the same workload scenario as in Figure 4.3, but with a QoS mechanism that balances memory performance by restricting the GPU to 8 outstanding requests and prioritizing all latency-sensitive CPU requests. When compared to Figure 4.3, the QoS mechanisms successfully prevent CPU starvation and CPU performance is not impacted by the GPU. With this static QoS, however, the GPU performance suffers. When `mcf` and `art` are run together with the GPU, the GPU receives barely enough bandwidth to maintain the frame rate. With the higher bandwidth `art-art` CPU workload, frame deadlines are missed.

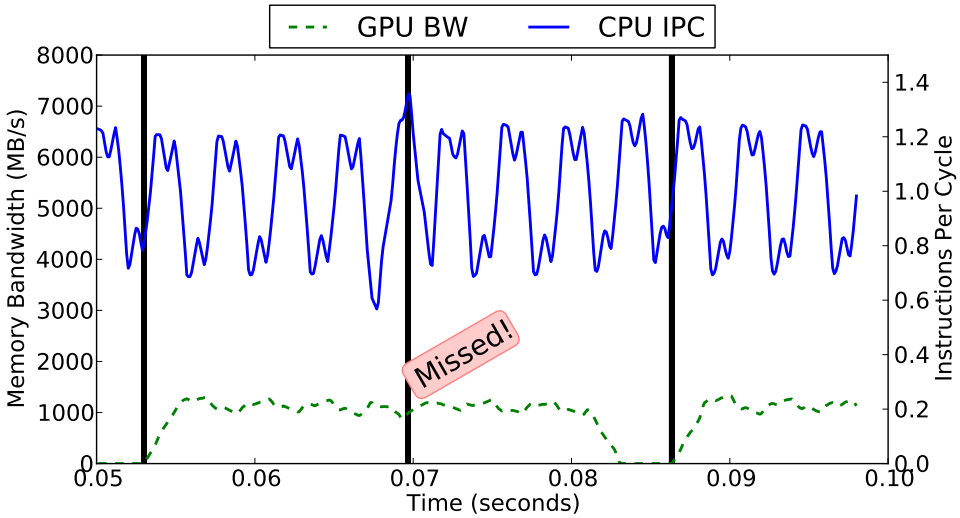
From this example and discussion, we can conclude that for the static QoS scheme to generally work for any workload scenario, the memory bandwidth needs to be over-provisioned for the worst case. Otherwise, it is possible that frames must be dropped, either while reconfiguring or while programming. A better alternative to costly over-provisioning, is to identify when the GPU should be allowed to nearly monopolize bandwidth to meet its real-time constraints, which I discuss in the next section.

4.2.2 Dynamic Quality of Service

Static QoS mechanisms lack the ability to adapt to the dynamic behavior of real workloads, resulting in either degraded CPU performance (Figure 4.3), or missed GPU deadlines (Figure 4.4(b)). In order to achieve high



(a) CPU cores: `mcf-art`, GPU out=8, CPU > GPU



(b) CPU cores: `art-art`, GPU out=8, CPU > GPU

Figure 4.4: GPU activity (bandwidth consumption) and CPU performance over time. GPU is restricted to at most 8 outstanding memory requests and CPU requests are given higher priority. Vertical lines represent frame deadlines. Experimental setup discussed in Section 4.3.

CPU performance while satisfying real-time constraints, I propose to dynamically adjust the QoS policy based on runtime workload characteristics. Ideally, CPU requests should be prioritized as long as the CPU does not compromise the GPU target frame rate. The key to achieving behavior that is near this ideal is

1. Identify when a deadline is likely to be missed
2. Only then adjust the QoS policy and either treat the GPU and CPU as equals, or even prioritize GPU requests.

We discuss how to predict when the GPU makes insufficient progress and a heuristic to adjust priority below.

4.2.2.1 Monitoring GPU Workload Progress

As discussed in Section 4.1.2, mobile GPUs typically partition the screen into equal-sized tiles and process them in order. Each tile is processed once for all primitives that overlap it, then it is not accessed again until the following frame. We exploit this to track the progress the GPU is making in the current frame. The GPU hardware is aware of how many tiles in total it must process, the order in which tiles are processed, and what tiles are currently active. Progress is thus simply the current position within the total frame, as described by Equation 4.1. This information can be easily obtained from the job manager unit, which keeps track of the tiles rendered.

$$FrameProgress = \frac{Number\ of\ tiles\ rendered}{Number\ of\ tiles} \tag{4.1}$$

Although this progress monitoring mechanism is simple, it is very effective in our system because the mobile GPU uses fine-grained tiles. The number of tiles is sufficiently large in the order of thousands and give good resolution to this progress metric. The variations in the complexity of tiles can affect the accuracy of this progress monitor, although we did not observe such complexity variation causing a problem in our evaluation. Such variations, however, can be averaged out by changing the order tiles are processed. Instead of processing tiles in a linear order from the top left of the screen, tiles distributed across the screen can be sequenced. With coarser tiles or non-tiled GPU architectures, a more sophisticated estimation of workload can be used, such as those suggested by prior work in the context of coarse-grained adjustments to the GPU voltage and frequency [22, 66] [1] .

4.2.2.2 Dynamic QoS Policy

To determine the QoS policy, the memory controller compares the frame progress rate, obtained above, with the *expected progress rate*. The expected progress rate can be calculated by dividing the time elapsed from the beginning

[1] Prior work estimates workload at frame granularity, and does not discuss monitoring in-frame progress dynamically. An additional hardware counter is needed to keep track in-frame progress, such as number of geometries processed. Our tile-based monitor does not need any additional hardware, as the tile bookkeeping is an integral part of the GPU's job management.

of the frame by the target frame time (e.g. $16.67ms$ for 60 frames-per-second (FPS)), as shown in Equation 4.2. As with tracking progress, more sophisticated techniques can be used to obtain higher accuracy estimates of expected progress [22], but this simple estimation was shown effective in our evaluations.

$$ExpectedProgress = \frac{Time\ elapsed\ in\ current\ frame}{Target\ frame\ time} \tag{4.2}$$

The memory controller then chooses a QoS policy based on how far the GPU is behind its expected progress point. Algorithm 1 shows an example dynamic QoS policy, which I employ in this paper and which works well in our experiments. There are two priority levels, and the CPU gets the higher priority as long as the current GPU progress rate is above the expected rate. When the progress falls behind the expected rate, GPU priority is increased to equal that of the CPU. When only 10% of the frame time remains until the deadline and if the GPU has not yet caught up to its expected point, the GPU is prioritized above the CPU in an attempt to make the frame deadline. This 10% buffer was chosen arbitrarily and can be tuned for better performance. Again, I favored a simple design that can demonstrate the benefits and importance of the dynamic approach. We leave refinements of this QoS selection algorithm to future work.

Figure 4.5(a) demonstrates how a static mechanism that prioritizes the CPU can lead to a missed GPU deadline. With our dynamic scheme, GPU priority is dynamically increased to enable it to meet its deadlines (Figure 4.5(b)).

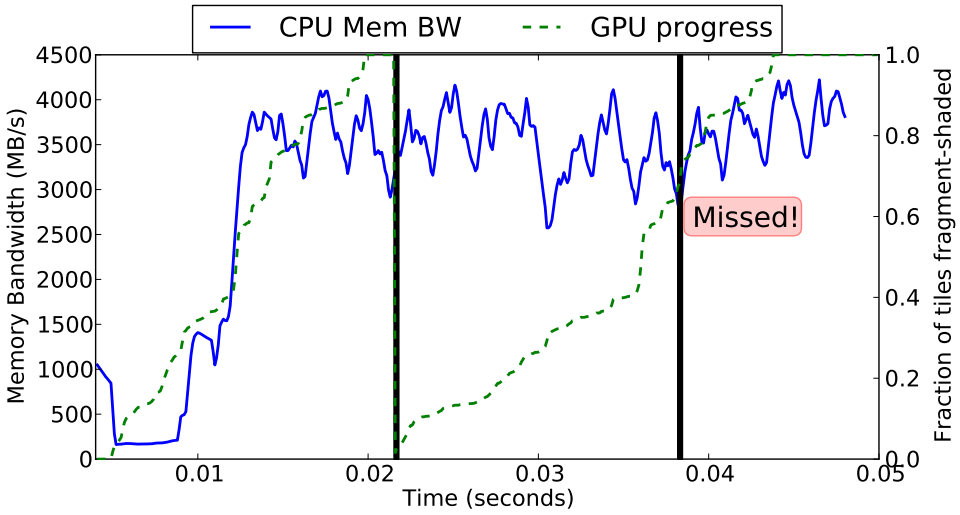
Algorithm 1 Dynamic QoS policy

if $FrameProgress > ExpectedProgress$ **then**
  $CPU_{priority} = High$
  $GPU_{priority} = Low$
else if $ExpectedProgress > 0.9$ **then**
  $CPU_{priority} = Low$
  $GPU_{priority} = High$
else
  $CPU_{priority} = GPU_{priority}$
end if

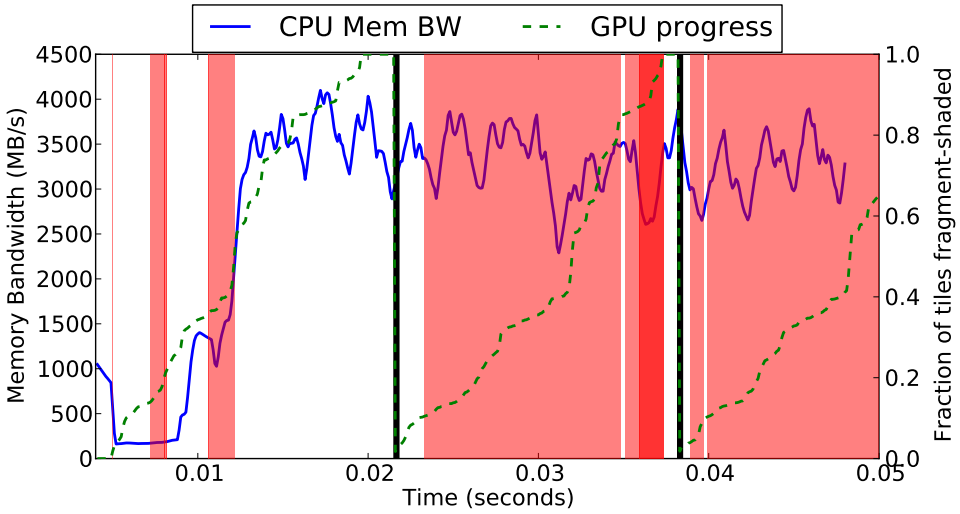
In the first frame, the GPU makes acceptable progress most of the time even with CPU priority. In the second frame, however, the GPU requires equal priority for much of the frame and higher priority towards the end of the frame to ensure the deadline is met.

4.3 Evaluation Methodology

I evaluate my proposed dynamic QoS scheme using cycle level simulations. I use a combination of the gem5 system simulator [17], a proprietary next-generation GPU simulator, and the DrSim DRAM simulator [31]. The gem5 out-of-order CPU model and the GPU model share the DRAM model through the gem5 bus. DrSim models memory controllers and DRAM modules faithfully, simulating the buffering of requests, scheduling of DRAM commands, contention on shared resources (such as address/command and data buses), and all latency and timing constraints of LPDDR2 DRAM.



(a) Static QoS leading to missed deadline



(b) Dynamic QoS adjusting priority to make deadline

Figure 4.5: CPU memory bandwidth consumption and GPU progress over several frames for static and dynamic QoS. Time intervals shaded in light red indicate times that GPU progress was insufficient and CPU and GPU priority are equal. Dark red shading indicates the critical periods of time that the dynamic scheme prioritizes GPU requests over CPU accesses.

Table 4.1: Simulated system parameters

CPU	Dual-core, 1.2GHz ARM out-of-order superscalar
Caches	32KB private L1 I/D, 1MB shared L2
GPU	8 Unified shader cores, 600MHz
GPU L2	128KB shared
System bus	128-bit wide, 1GHz
Memory controller	FR-FCFS scheduling, open row policy 64 entries read queue, 64 entries write queue
Main memory	1 channel, 1 rank / channel, 8 banks / rank 4 x16 LPDDR2-1066 chips / rank 8.3GB/s peak BW, All chip parameters from the latest Micron datasheet [51] XOR-interleaved bank index [77]

System configuration

The QoS schemes I simulate include uncontrolled CPU and GPU (`noqos`), static CPU priority over GPU (`static`), and our dynamic scheme (`dynamic`). Constraining the number of outstanding GPU requests to N (`outN`) is used in combination with `static` and `dynamic`.

Table 4.1 summarizes the parameters of our simulated systems. I believe the simulated system is representative of the next-generation high-end mobile SoC. Memory scheduling queues are large enough to guarantee room for CPU requests even when GPU was not constrained in `noqos`.

Workloads

Due to the slow GPU simulation speed, it is impractical to run a GPU accelerated application and other memory intensive applications on top of the

full OS and GPU driver stack. Instead, I run CPU workloads on the CPU cores in parallel with graphics workloads on the GPU to approximate the memory bandwidth constrained usage scenario.

Table 4.2 shows the CPU and the GPU benchmarks used. I selected two SPEC CPU 2000 benchmarks with the MinneSPEC input set [40], which place significant demand on the memory system. Dual-core CPU workloads are multi-programmed to simulate three levels of CPU memory bandwidth usage. GPU workloads are post-driver output of a representative frame from each graphics benchmark; `taiji` and `egypt` are WVGA resolution and `taiji1080p` and `farcry` are 1080p. Their target performance in frames-per-second (FPS) was determined by measuring the their execution time on GPU without CPU interference. Two workloads, `taiji1080p` and `farcry` were not able to finish in 16.67ms on our simulated system, but finished within 33.34ms. I assume that missed frames are skipped and the behavior is repetitive, so the target FPS is an integer divisor of the 60 FPS base.

4.4 Results

In this section I present experimental results that demonstrate the effectiveness of the dynamic mechanism. I focus on challenging workloads that are constrained by the available memory bandwidth of the SoC. In such cases, it is impossible to simultaneously meet the target frame rate and service the CPU without GPU interference. I analyze the interaction between the components and show how dynamic can adapt to the changes in workload demand com-

Table 4.2: Workloads used and their characteristics.

GPU workload	Source	Target FPS
`taiji`	3DMarkMobile ES 2.0 [4]	60
`egypt`	GLBenchmark [39]	60
`taiji1080p`	3DMarkMobile ES 2.0	30
`farcry`	Game	30
CPU workload	**Average Mem Bandwidth**	
`art-art`		5.8GB/s
`mcf-art`		3.5GB/s
`mcf-mcf`		800MB/s

bination and provide the near-optimal QoS strategy, while current guidelines for static QoS fail.

To better quantify these interactions, I show the results of multiple QoS schemes for GPU and CPU performance in Figure 4.6 and Figure 4.7, respectively. The results point to two important insights, which contradict the current best practice of prioritizing the CPU and constraining the GPU.

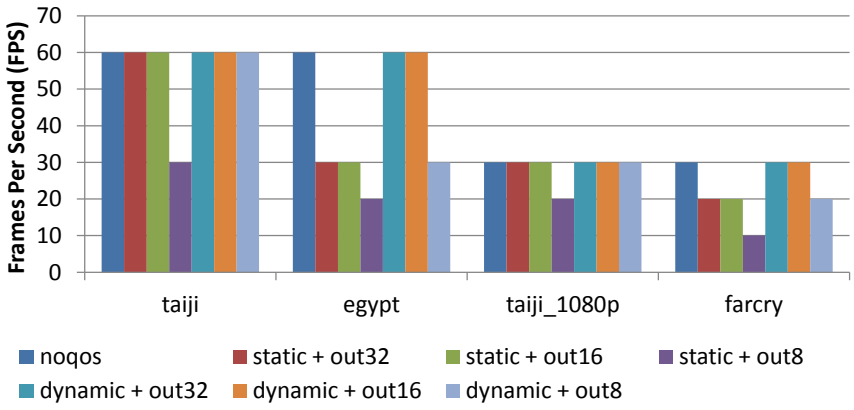


Figure 4.6: GPU performance in frames per second (FPS) when both CPU cores run `art`, the most bandwidth demanding CPU workload.

First, restricting the number of outstanding GPU requests can cause the GPU to miss deadlines, while at the same time often *degrading* CPU performance. As shown in Figure 4.6, restricting the GPU to 8 outstanding requests reduces the frame rate by 33% or 50%, dropping one of every three or two frames respectively. The impact on the CPU is interesting. Figure 4.7 shows that as long as the GPU meets its deadlines (configurations for which the GPU fails are shaded black in Figure 4.7) the CPU either sees little benefit from constraining the GPU, or experiences noticeable performance degradation. I found that having fewer GPU requests for the memory scheduler to choose from prevents efficient scheduling and reduces the effective memory bandwidth. Both the GPU and CPU cores suffer from the longer low effective bandwidth period.

The CPU benefits from a constrained GPU only when the GPU fails to meet its required frame rate (black bars). For example, restricting the GPU to 8 outstanding accesses leads to only a 5% degradation in CPU performance in `farcry-art-art`. The GPU however, only achieves 10 FPS instead of the targeted 30 FPS. This is generally unacceptable.

Second, our GPU progress-aware `dynamic` QoS mechanism can indeed adapt to the changes in CPU and GPU workloads and provide the performance of the best QoS setting for each workload. When bandwidth is not sufficient for the workloads (`egypt` and `farcry` in Figure 4.6), only `noqos` and `dynamic` meet GPU performance requirements. Even in such severely bandwidth-constrained cases, `dynamic` can still find opportunity to give the CPU some priority and

reduces slowdown by 3.6% and 6.9% from `noqos`. When there is sufficient bandwidth to serve the GPU and CPU cores simultaneously, CPU performance of `dynamic` + `out32` roughly matches the best `static` configuration of each workload (shown as the lowest non-black bar in Figure 4.7) and provides up to 9.4% reduction in slowdown from `noqos`.

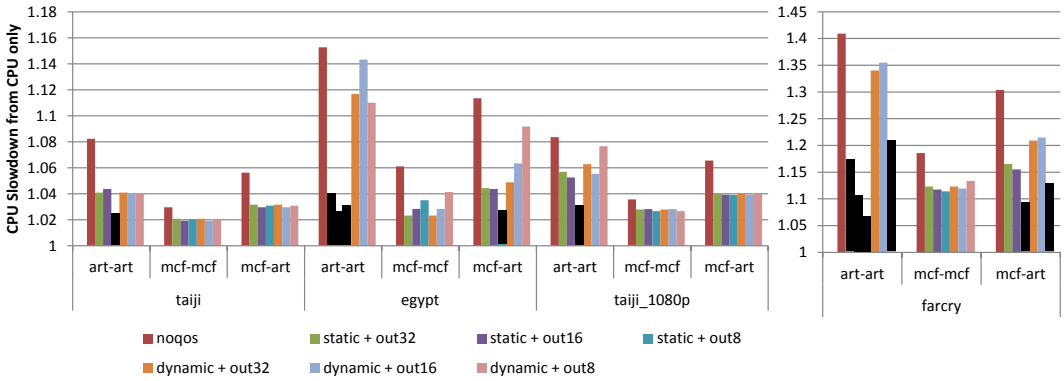


Figure 4.7: Slowdown of the CPU relative to its performance in an SoC with the same memory configuration but no GPU. Black bars represent configurations in which the GPU could not meet the workload's performance target.

Again, constraining the outstanding number of requests from the GPU hurts CPU performance even more for `dynamic` with `egypt` and `mcf-art`. It turns out that the constraint slows down GPU progress and our `dynamic` scheme forces the memory controller to raise GPU priority, and therefore the CPU suffers. Since `mcf-art` uses a moderate amount of memory bandwidth, GPU requests can be issued opportunistically even with low priority. Therefore, having many outstanding requests at the memory controller ready allows them to be scheduled efficiently and enables the GPU to make good progress. In the `dynamic` + `out32` configuration, the memory controller doesn't raise

GPU priority and the CPU gets low-latency priority accesses, yielding better performance.

4.5 Related Work

4.5.1 QoS

Tune and Bruce [71] surveyed standard MPSoC QoS mechanisms, such as regulating the number of outstanding requests, regulating traffic injection rate, and prioritizing CPU requests. Using a trace-based simulation of a system consisting of two CPUs, a GPU, and a LCD controller, they showed that static QoS mechanisms can be used to control bandwidth allocation. In their target system, however, memory bandwidth was not a constraint in their system; the workloads did not cause any missed GPU deadlines. Such a configuration is rare given the increasing sophistication of GPU and CPU cores.

4.5.2 Memory Controller Design for Heterogeneous SoCs

Akesson et al. [12] describes a memory controller for a real-time systems in which there is a separate request queue for each master. The scheduler then arbitrates between different queues based on the bandwidth allocated to each master. Ausavarungnirun et al. [15] describes the similar motivation for having multiple request queues. They identified that previously proposed fairness-aware scheduling algorithms require a large request queue to work because bandwidth intensive GPU can fill up a large fraction of the queue. In this chapter, I instead prevented GPU cores flooding the request queue by lim-

iting the number of outstanding requests from the GPU. They are equivalent mechanisms in ensuring enough queue slots for CPU cores.

4.5.3 GPU DVFS

Several previous studies proposed dynamic voltage and frequency scaling (DVFS) for graphics to reduce GPU power consumption. The integral component of GPU DVFS is estimating the GPU workload complexity to determine the voltage-frequency level that consumes the least amount of power while meeting the frame deadlines. Similarly, shared-memory QoS also has to figure out the appropriate QoS level for GPU that meets frame deadlines while minimizing the impact to the CPU performance.

Wang et al. [60] evaluate a technique to dynamically power-gate GPU shader processing elements. They predict the amount of work in the current frame using a history of N prior frames. Silpa et al. [66] determine the DVFS level at a frame boundary by estimating frame complexity through workload analysis. They compute complexity from frame-specific parameters such as the number of vertices, vertex shader program instruction count, number of primitives, total area covered by all primitives, and the number of textures used. Gu et al. [22] combines both styles of estimating workload: current workload parameters obtained from the data structures describing a frame; and the prediction of a proportional-integral-derivative (PID) controller based on the observed execution time of previous frames.

Unlike adjusting memory scheduling priority, the overhead of changing DVFS levels is very high, requiring both significant time and energy. Therefore, these DVFS techniques evaluated workload complexities on frame boundaries. QoS for shared memory system, on the other hand, can track progress at much finer granularity to make very dynamic decisions on scheduling policy. The proposed progress monitor is simple to integrate with current mobile GPUs, does not store any history, and requires no sophisticated computation or analysis.

4.6 Summary

In this chapter, I carefully analyzed the performance of a system that is representative of current and upcoming advanced SoCs. I used a cycle-level simulator that accurately models an SoC with two CPU cores and a mobile GPU, which all share a single DRAM main memory system. By evaluating the complex interactions between these components I show that current best-practice QoS mechanisms are insufficient and often apply the wrong QoS policy. I showed that there is no single static policy that can be used for workloads with varying demands to simultaneously meet the requirements of the GPU without significantly impacting the CPU cores.

I use this insight to develop a dynamic QoS scheme that maintains CPU priority when possible, but shifts priority towards the GPU if it predicts that the GPU will miss a real-time deadline. I propose a simple, yet effective, tile based frame progress tracking mechanism to enable dynamic QoS policy

decisions and show that it both enables the GPU to meet its deadlines and minimizes impact on the CPU. Using this technique, I also conclude that restricting the number of outstanding GPU requests, a static QoS mechanism in use today, often degrades the performance of all cores, because it limits the ability of the memory scheduler to exploit locality.

Chapter 5

Conclusions and Future Research Directions

Future systems will be increasingly parallel and heterogeneous. These changes in processor design also change the way the processors use off-chip bandwidth. Memory systems developed for unicore processors are inefficient in handling accesses from such heterogeneous parallel processors. In this dissertation, I presented how we can include parallelism and heterogeneity as a first class design consideration for the memory systems of future systems. Specifically, I explored to broad techniques. The first adapts DRAM resource allocation to balance parallelism and locality and the second dynamically adjust priority to balance the diverse QoS requirements present in heterogeneous systems. I summarize the main insights and conclusions based on these two lines of research below.

Independent threads running on multiple cores in parallel share a DRAM system that has complex structure with multiple channels, ranks, and banks. Traditionally, all these DRAM resources have been uniformly shared by the cores which results in severe interference between threads running concurrently. However, a thread benefits very little from having an access to all the resources because one core typically cannot utilize that much parallelism. I

demonstrate that by carefully controlling which subset of DRAM resources each thread accesses, we can control interference as well as getting most of the benefits a core can attain from parallelism. I describe how the OS can manage the allocation of DRAM resources as an optimization to its frame allocation algorithm. Utilizing the OS has the advantage of greater flexibility and a simpler path to adoption. By localizing spatially adjacent accesses to a small number of resources (e.g., banks), efficient page-mode accesses are possible and interference is reduced. By judiciously distributing accesses of a thread across resources (e.g., banks and channels), conflicting long-latency requests can be overlapped. I explain that is this balance between these two that enables efficient sharing of the memory system between multiple cores.

Experimental data suggests that bandwidth-limited CMP systems can benefit more from exploiting locality in each thread than from providing memory parallelism to each thread. With increasing number of cores and slower off-chip memory bandwidth scaling, the higher efficiency brought by localization becomes more critical in achieving high bandwidth utilization. Localization also reduces the number of banks and channels each thread accesses simultaneously. This reduces inter-thread structural conflicts and locality interference. My evaluation shows that preserving locality of applications with bank-partitioning significantly improves system throughput, by up to 18% in the configuration I tested. Even at the level of memory channels, localizing accesses, enabled by coarse-grained interleaving, is shown to improve the throughput of a CMP processor as a whole. The bandwidth of a single mem-

ory channel has been increasing steadily whereas the performance of a single thread has mostly stagnated. As a result, the gain from the reduction of interferences outweighs the loss from a smaller per-thread share of channel bandwidth. For a high-performance out-of-order core, 8-16 banks / core are typically sufficient to provide all the benefits of bank-level parallelism. When the number of banks per thread after partitioning falls shorts to this amount, we can restructure the DIMM to increase independent banks with subranking. Because we already have many parallel threads running that can make use of parallelism in the memory system, each thread is better off exploiting its locality than oversubscribing memory with additional parallelism.

I believe we can continue to exploit the locality in access streams for future CMP systems. The techniques I suggest in this dissertation successfully exploit locality and achieve good performance and fairness for current systems with realistic memory system parameters. As off-chip bandwidth becomes scarcer, the importance of locality, which enables efficient utilization of memory bandwidth, grows as well. Even in future many-core processors for which the number of threads may exceed the number of banks, locality can still be exploited. With a large number of cores, it is likely that groups of cores will run parallel threads that cooperate, share data, and make roughly synchronized memory accesses. Examples of such an architecture are current GP-GPUs. The principle of preserving locality within each coordinated access stream for a group of cores and reducing interference between multiple access streams applies just the same.

Exploiting the differences in memory access characteristics can lead to better system performance under limited off-chip memory bandwidth constraints. Applications running concurrently in heterogeneous cores have various degrees of latency sensitivity, bandwidth requirements, and deadline requirements. Such differences make the same bandwidth resource be much more useful for some particular cores than for others, at different times. Hence, how critical resources are to different cores changes dynamically. Statically partitioning the limited resources between different types of cores without considering such criticality requirements is not optimal. It either results in an over-provisioning of resources or missed applications requirements when the workload mix is significantly different from that which the static design targeted.

Almost all SoCs for consumer products now include powerful CPU and GPU cores with very different requirements that can benefit from the requirement-aware management principle. My evaluation shows that by prioritizing the latency-sensitive CPU requests, CPU performance can be improved without hurting GPU frame rate as long as deadlines are met. There is no single static policy that can satisfy workloads with varying demands to simultaneously meet the requirements of the GPU and not significantly sacrifice CPU core performance. I propose a dynamic QoS scheme that does achieve the goal of meeting all deadlines while maximizing possible CPU performance. This QoS mechanism maintains CPU priority when possible, but shifts priority toward the GPU if it predicts that the GPU will miss a real-time deadline.

I used a GPU and CPU as an example of cores with conflicting demands: latency-sensitive best-effort CPU cores and a bandwidth-sensitive real-time GPU. These diverse requirements are common and a growing number of cores share an increasingly constrained memory system. I believe dynamic techniques, such as the one presented, are the key to enabling such future systems to meet user requirements while still efficiently utilizing scarce shared resources. Thus my conclusions are important and open the way to additional research.

5.1 Future Research Directions

There are several opportunities for short-term future work that can extend the topics discussed in this dissertation.

Real system implementation: We can combine the locality-aware memory allocation and dynamic quality-of-service proposed in real systems. Real system implementation will enable studying a broader range of applications, including 3D-games for an extended period of execution time. Also, the CPU and GPU applications have very different memory access characteristics, hence can benefit from careful data placement in DRAM. To test such a system, both bank partitioning and dynamic QoS are required. With a technique presented by Liu et al. [47] that reverse-engineers the hard-wired bank-index hashing function, we can implement bank-partitioning in software for real hardware. I have already invested effort to get an ARM SoC-based board working with run-time access to the GPU performance counter which can be used to track

the frame progress (the number of write-backs for tile). The particular SoC (Samsung Exynos) I worked on lacked QoS control support, such as memory controller priority and number of outstanding requests. Newer SoCs have interconnect and memory controller IP that supports request priority [7, 8] which we can use to implement dynamic QoS policy.

Adaptive bank-partitioning: Multi-programmed workloads change as applications start and finish their execution. In this dissertation, I evaluated a scenario where all cores are active and banks are uniformly partitioned among those applications running on the cores. When only a subset of cores are active, we can reassign the banks of the inactive cores to active cores for additional bank-level parallelism or capacity. Later, when a new application starts executing on an inactive core, we either lose isolation or have to migrate the pages to the original partition. The benefit of bank-borrowing and the cost of page migration can be quantitatively measured to design an intelligent run-time allocation policy.

We can also adapt the address mappings of each application based on its memory access characteristics, such as locality. For example, low locality applications can use more banks than high locality applications. I have evaluated this non-uniform bank partitioning based on locality. Interestingly, even high locality benchmarks that exhibit a sequential access pattern (ex. `libquantum`), still require a modest number of banks because of write-back traffic, which has low locality because of unpredictable cache replacement. The access pattern for the write-backs are rather random regardless of the

locality of applications because write-backs to DRAM occur when the cache-lines are evicted, not when the store instructions are executed. Techniques that proactively write-back dirty lines, such as Eager Writeback [45] or the Virtual Write Queue [70], can improve locality of writes and expected to mitigate such asymmetry in read-write locality.

Interactions between CMP Scheduling and Address Mapping: A number of memory controller scheduling policies for shared-memory CMP systems, which aim to improve fairness and system throughput, have been proposed [35, 37, 38, 54, 55, 58, 61]. As discussed in Section 3.6, bank-partitioning addresses these issues in a fundamentally different way. It does not introduce a new scheduling technique and instead resolves the fairness issue arising from a high locality thread delaying low locality threads by disallowing both to occupy the same bank. Similarly to Kaseridis et al. [35], we can apply various thread-aware memory scheduling algorithms on top of the bank-partitioning idea to further improve fairness of CMP systems. Since bank-partitioning restores locality, it improves high locality applications more than it does low locality benchmarks. Therefore, I believe a fairness-aware scheduling algorithm which provides fair memory service to low locality benchmarks will complement bank-partitioning very well.

Bibliography

[1] Intel 82955x memory controller hub (mch) datasheet. Technical Report 306828-001, Intel, Oct. 2005.

[2] Calculating memory system power for DDR3. Technical Report TN-41-01, Micron Technology, 2007.

[3] Ibm power 795 technical overview and introduction. Technical Report REDP-4640-00, IBM, September 2010.

[4] 3DMarkMobile ES 2.0. `http://www.futuremark.com/products/3dmarkmobile`, 2011.

[5] ITRS 2011 update. Technical report, International Technology Roadmap for Semiconductors, 2011.

[6] 3rd generation intel core processor family and intel pentium processor family datasheet. Technical Report 306828-001, Intel, Sep. 2012.

[7] Corelink cci-400 cache coherent interconnect. Technical Report ARM DDI 0470G (ID111612), ARM, 2012.

[8] Corelink dmc-400 dynamic memory controller. Technical Report ARM DDI 0466C (ID072812), ARM, 2012.

[9] Cuda c best practices guide. Technical Report DG-05603-001, Nvidia, 2012.

[10] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber. Future scaling of processor-memory interfaces. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 42:1–42:12, New York, NY, USA, 2009. ACM.

[11] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi. Multicore dimm: an energy efficient memory module with independently controlled drams. *IEEE Compututer Architecture Letters*, 8(1):5–8, Jan. 2009.

[12] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 251–256, New York, NY, USA, 2007. ACM.

[13] B. Akesson, P.-C. Huang, F. Clermidy, D. Dutoit, K. Goossens, Y.-H. Chang, T.-W. Kuo, P. Vivet, and D. Wingard. Memory controllers for high-performance and real-time mpsocs: requirements, architectures, and future trends. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '11, pages 3–12, New York, NY, USA, 2011. ACM.

[14] J. Alakarhu and J. Niittylahti. A comparison of precharge policies with modern dram architectures. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, volume 2, pages 823 – 826 vol.2, 2002.

[15] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 416–427, Piscataway, NJ, USA, 2012. IEEE Press.

[16] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88 –598, feb. 2008.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39:1–7, Aug. 2011.

[18] T. M. Brewer. Instruction set innovations for the convey hc-1 computer. *IEEE Micro*, 30(2):70–79, Mar. 2010.

[19] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the nineteenth annual ACM symposium on Parallel algo-*

rithms and architectures, SPAA '07, pages 105–115, New York, NY, USA, 2007. ACM.

[20] E. Cooper-Balis, P. Rosenfeld, and B. Jacob. Buffer-on-board memory systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 392–403, Piscataway, NJ, USA, 2012. IEEE Press.

[21] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM.

[22] Y. Gu and S. Chakraborty. A Hybrid DVS Scheme for Interactive 3D Games. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12. IEEE, Apr. 2008.

[23] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[24] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[25] J. Huh, D. Burger, and S. Keckler. Exploring the design space of future cmps. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 199 –210, 2001.

[26] I. Hur and C. Lin. A comprehensive approach to dram power management. In *High Performance Computer Architecture (HPCA), 2008 IEEE 14th International Symposium on*, pages 305 –316, feb. 2008.

[27] Ibm. Ibm ddr3 memory for system x. `http://ark.intel.com/products/codename/1791/Prescott`.

[28] Intel. Products (formerly prescott). `http://ark.intel.com/products/codename/1791/Prescott`.

[29] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.

[30] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *Proceedings of the 49th annual Design Automation Conference*, DAC '12, New York, NY, USA, 2012. ACM.

[31] M. K. Jeong, D. H. Yoon, and M. Erez. DrSim: A platform for flexible DRAM system research. `http://lph.ece.utexas.edu/public/DrSim`.

[32] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing dram locality and parallelism in shared memory cmp systems.

In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, February 2012.

[33] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues*, 2002.

[34] D. Kaseridis, J. Stuecheli, J. Chen, and L. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –11, jan. 2010.

[35] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist open-page: a dram page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 24–35, New York, NY, USA, 2011. ACM.

[36] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. *SIGARCH Comput. Archit. News*, 37(3):140–151, June 2009.

[37] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, Jan 2010.

[38] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 65–76, Washington, DC, USA, 2010. IEEE Computer Society.

[39] Kishonti Informatics Ltd. GLBenchmark. `http://www.glbenchmark.com`, 2011.

[40] A. J. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *IEEE Computer Architecture Letters*, 1:7–, January 2002.

[41] L. Kurian, B. Choi, P. T. Hulina, and L. D. Coraor. Module partitioning and interlaced data placement schemes to reduce conflicts in interleaved memories. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*, ICPP '94, pages 212–219, Washington, DC, USA, 1994. IEEE Computer Society.

[42] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. *SIGOPS Oper. Syst. Rev.*, 34(5):105–116, Nov. 2000.

[43] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IX, pages 105–116, New York, NY, USA, 2000. ACM.

[44] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware dram controllers. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 200–209, Washington, DC, USA, 2008. IEEE Computer Society.

[45] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proc. the 33rd IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, Nov.-Dec. 2000.

[46] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 315–326, Washington, DC, USA, 2008. IEEE Computer Society.

[47] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 367–376, New York, NY, USA, 2012. ACM.

[48] G. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 53 –64, april 2009.

[49] M. McTague and H. David. Fully buffered DIMM (FB-DIMM) design considerations. Intel Developer Forum (IDF), Feb. 2004.

[50] W. Mi, X. Feng, J. Xue, and Y. Jia. Software-hardware cooperative DRAM bank partitioning for chip multiprocessors. In *Proceedings of the 2010 IFIP international conference on Network and parallel computing*, NPC'10, pages 329–343, Berlin, Heidelberg, 2010. Springer-Verlag.

[51] Micron Corp. *Micron 2 Gb* ×16*,* ×32*, Mobile LPDDR2 SDRAM S4*, 2011.

[52] Micron Corp. *Micron 2 Gb* ×4*,* ×8*,* ×16*, DDR3 SDRAM: MT41J512M4, MT41J256M4, and MT41J128M16*, 2011.

[53] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 374–385, New York, NY, USA, 2011. ACM.

[54] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society.

[55] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Pro-*

ceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

[56] P. A. Navrátil, D. S. Fussell, C. Lin, and W. R. Mark. Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. In *IEEE/EG Symposium on Interactive Ray Tracing 2007*, pages 95–104. IEEE/EG, Sep 2007.

[57] U. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. Kumar, and H. Park. An 8-core 64-thread 64b power-efficient sparc soc. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 108 –590, feb. 2007.

[58] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222. IEEE Computer Society, 2006.

[59] V. K. Pingali, S. A. McKee, W. C. Hseih, and J. B. Carter. Computation regrouping: restructuring programs for temporal data cache locality. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 252–261, New York, NY, USA, 2002. ACM.

[60] Y.-M. C. Po-Han Wang. A Predictive Shutdown Technique for GPU Shader Processors. *IEEE Computer Architecture Letters*, 8(1):9–12, Jan. 2009.

[61] N. Rafique, W. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 245–258. IEEE Computer Society, 2007.

[62] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[63] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 371–382, New York, NY, USA, 2009. ACM.

[64] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[65] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 45–57, New York, NY, USA, 2002. ACM.

[66] B. Silpa, G. Krishnaiah, and P. R. Panda. Rank based dynamic voltage and frequency scaling for tiled graphics processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 3–12, New York, NY, USA, 2010. ACM.

[67] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IX, pages 234–244, New York, NY, USA, 2000. ACM.

[68] Standard Performance Evaluation Corporation. SPEC CPU 2006. `http://www.spec.org/cpu2006/`, 2006.

[69] A. Stevens. Qos for high-performance and power-efficient hd multimedia. Technical report, Arm, 2010.

[70] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The virtual write queue: coordinating dram and last-level cache policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 72–82, New York, NY, USA, 2010. ACM.

[71] A. Tune and A. Bruce. How to tune your SoC to avoid traffic congestion. In *DesignCon*, 2010.

[72] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi. Rethinking dram design and organization for energy-constrained multi-cores. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 175–186, New York, NY, USA, 2010. ACM.

[73] F. A. Ware and C. Hampel. Improving power and data efficiency with threaded memory modules. In *Proceedings of International Conference on Computer Design*, 2006.

[74] P. Washkewicz. Ddr3 memory buffer: Buffer at the heart of the lrdimm architecture. Technical report, Inphi.

[75] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive granularity memory systems: a tradeoff between storage efficiency and throughput. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 295–306, New York, NY, USA, 2011. ACM.

[76] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez. The dynamic granularity memory system. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 548–559, Piscataway, NJ, USA, 2012. IEEE Press.

[77] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on*

Microarchitecture, MICRO 33, pages 32–41, New York, NY, USA, 2000. ACM.

[78] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive dram architecture for improving memory power efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 210–221, Washington, DC, USA, 2008. IEEE Computer Society.

[79] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu. Decoupled dimm: building high-bandwidth memory system using low-speed dram devices. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 255–266, New York, NY, USA, 2009. ACM.

[80] H. Zheng and Z. Zhu. Power and performance trade-offs in contemporary dram system designs for multicore processors. *Computers, IEEE Transactions on*, 59(8):1033 –1046, aug. 2010.

Vita

Min Kyu Jeong was born in Incheon, Korea on November 12, 1982. Min Kyu studied at Seoul National University, Seoul, Korea, where he received the degree of Bachelor of Science in Computer Science and Engineering in February 2006.

Min Kyu started his graduate study on computer architecture at the University of Texas at Austin in 2006, and received an M.S. degree in Electrical and Computer Engineering in 2008. His research is focused on memory systems, including caches, DRAM, and emerging non-volatile memory. His research has been published in major computer architecture conferences such as ISCA, HPCA, and DAC. Min Kyu worked as a research intern at Nvidia, Intel, and ARM throughout his graduate student career.

Permanent address: 13 Loftus St. Campsie, NSW 2194 Australia

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.